



High-Performance Web Apps with FastAPI

The Asynchronous Web Framework
Based on Modern Python

—
Malhar Lathkar

Apress®

High-Performance Web Apps with FastAPI

**The Asynchronous
Web Framework Based on
Modern Python**

Malhar Lathkar

Apress®

High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python

Malhar Lathkar

Nanded, Maharashtra, India

ISBN-13 (pbk): 978-1-4842-9177-1

ISBN-13 (electronic): 978-1-4842-9178-8

<https://doi.org/10.1007/978-1-4842-9178-8>

Copyright © 2023 by Malhar Lathkar

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: Divya Modi

Development Editor: James Markham

Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Piaxyby

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at <https://github.com/Apress/Build-High-Performance-Web-Apps-with-FastAPI-by-Malhar-Lathkar>. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Dedicated to my late father,

Shri V. K. Lathkar.

You stood behind me like a rock.

It helped me to get rid of fear of failure.

Table of Contents

About the Authorxiii

About the Technical Reviewerxv

Acknowledgmentsxvii

Introductionxix

Chapter 1: Introduction to FastAPI 1

 Type Hints 2

 The typing Module 6

Asynchronous Processing 8

 The asyncio Module 8

 ASGI 11

About REST Architecture 13

 What Is an API? 14

 REST 17

 REST Constraints 17

HTTP Verbs 19

 POST Method 19

 GET Method 20

 PUT Method 20

 DELETE Method 21

FastAPI Dependencies 21

 Starlette 22

TABLE OF CONTENTS

Pydantic.....	23
Uvicorn	23
Installation of FastAPI	24
Summary.....	28
Chapter 2: Getting Started with FastAPI.....	29
Hello World.....	30
Create an Application Object	30
Path Operation Decorator	30
Path Operation Function	32
Start Uvicorn.....	32
Externally Visible Server	34
Interactive API Docs	36
Swagger UI	37
Redoc	42
JSON Schema.....	44
Path Parameters	46
Using Type Hints	47
Type Parsing	48
Query Parameters	49
Optional Parameters.....	51
Order of Parameters	53
Validation of Parameters.....	55
Validating String Parameter.....	56
Validation with RegEx.....	59
Validating Numeric Parameters.....	60
Adding Metadata	62
Summary.....	64

Chapter 3: Request Body	65
POST Method	65
Body Parameters.....	67
Data Model with Pydantic	69
dataclasses Module.....	70
BaseModel.....	71
Pydantic Model As Parameter	73
Model Configuration.....	76
orm_mode	77
Pydantic Fields.....	80
Validation	82
Custom Validation.....	85
Nested Models	87
Summary.....	91
Chapter 4: Templates.....	93
HTML Response	93
Template Engine	96
Hello World Template	98
Template with Path Parameter.....	100
Template Variables.....	101
Passing dict in Template Context.....	102
Conditional Blocks in Template.....	103
Loop in Template	105
Serving Static Assets	107
Using JavaScript in Template	108
Static Image	111
CSS As a Static Asset	113

TABLE OF CONTENTS

HTML Form Template	115
Retrieve Form Data	117
Summary.....	120
Chapter 5: Response.....	121
Response Model.....	122
Cookies	125
set_cookie() Method.....	125
Cookie Parameter	126
Headers.....	129
Header Parameter	130
Response Status Code	131
Response Types	134
HTMLResponse.....	135
JSONResponse	136
StreamingResponse	136
FileResponse	138
RedirectResponse.....	139
Summary.....	141
Chapter 6: Using Databases.....	143
DB-API.....	144
Creating the Books Table	144
Inserting a New Book	147
Selecting All Books	150
Selecting a Single Book.....	151
Updating a Book	152
Deleting a Book	153

aiosqlite Module.....	154
SQLAlchemy.....	156
async in SQLAlchemy.....	163
databases Module	164
Core Expression Language	165
Table Class Methods.....	166
FastAPI Path Operations	167
PyMongo for MongoDB	170
Motor for MongoDB.....	177
Summary.....	179
Chapter 7: Bigger Applications.....	181
Single File App	182
APIRouter	185
Router Package	189
Mounting Subapplications	191
Dependencies	194
Example of Dependency Injection	195
Query Parameters As Dependencies	196
Parameterized Dependency Function.....	200
Using Class As Dependency	202
Database Session Dependency	204
Dependency in Decorator	205
Middleware	207
CORS	209
Summary.....	210

TABLE OF CONTENTS

Chapter 8: Advanced Features.....211

 WebSockets 211

 How Do WebSockets Work? 212

 WebSocket Server 213

 WebSocket Client 214

 WebSockets Module in FastAPI..... 215

 Test WebSockets with Insomnia 221

 Multiclient Chat Application 222

 GraphQL 226

 The Schema Definition Language..... 227

 Queries 228

 Mutations..... 229

 Subscriptions..... 230

 Schema..... 231

 Strawberry GraphQL 231

 FastAPI Events 238

 Mounting WSGI Application 239

 Summary..... 241

Chapter 9: Security and Testing.....243

 Exception Handling 243

 User-Defined Exception 245

 Security..... 248

 Basic Access Authentication..... 248

 OAuth 250

 OAuth2PasswordBearer..... 252

TABLE OF CONTENTS

Testing.....	260
Testing WebSocket	263
Testing Databases	265
AsyncClient	270
Summary.....	272
Chapter 10: Deployment	273
Hercorn	274
HTTPS	275
Daphne	277
Gunicorn.....	278
FastAPI on Render Cloud.....	279
Docker.....	282
Google Cloud Platform	285
Deta Cloud.....	289
Summary.....	293
Index.....	295

About the Author



Malhar Lathkar is an independent developer, trainer, technical writer, and author with over 30 years of experience. He holds a postgraduate degree in electronics. After a brief stint as a degree college lecturer, he entered into the software training and development field as an entrepreneur.

Malhar is mostly a self-taught professional. Over the years, he has gained proficiency in various programming technologies and guided thousands of students and professionals from India and different countries around the world. Malhar also offers software training services to corporates.

He has been associated with many EdTech companies as a freelance content developer and subject matter expert. He has also written a few books that have been published by well-known publishing houses.

Malhar is frequently invited to conduct workshops, deliver technical talks for the students in various engineering colleges, and work as a jury to evaluate student projects for hackathon competitions.

He enjoys Indian classical music. Being an avid sportsman during college days, he keeps a keen eye on all the sporting action around the world.

About the Technical Reviewer



Jeff Chiu is a senior software engineer with over ten years of experience working on Django, Python, and REST APIs. He has worked as senior engineer at several major Silicon Valley tech companies building platform infrastructure. Jeff writes clean, consistent code. Outside of work, he mentors other aspiring engineers and early career professionals through the online community.

He enjoys this so much that he has built multiple apps and created discussion forums to help engineers receive constructive feedback. His work portfolio can be found at <https://jeffchiucp.github.io/portfolio/>.

Acknowledgments

At the outset, I express my sincere gratitude toward Apress (Springer Nature) Publications for giving me this opportunity to write this book and be a part of the Apress family. I thank the editorial team and especially Jeff Chiu – the technical reviewer – for his invaluable inputs while finalizing the draft of this book.

I would also like to acknowledge the graphics designers who have produced a splendid cover page for this book.

The unerring and unconditional support of my family (my wife Jayashree, daughter Sukhada, and son-in-law Shripad) in my endeavors has always been my biggest strength. They have stood by me in good and bad times. A very dear friend Dr. Kishore Atnurkar and his wife Seema, who are no less than a part of my family, have been appreciative of my work and have always given me a lot of encouragement. It wouldn't be out of place to acknowledge their contribution.

Throughout my academic life, I have been blessed with guidance from some highly inspiring teachers. Their profound influence has made me a lifelong learner. I hereby pay my respectful regards to all my teachers.

You always learn more when you teach. I would like to thank thousands of my students for being a part of my learning journey.

Finally, for all those who have been involved in bringing out this book, a big thank you!

Introduction

As a programming language, Python has been continuously evolving. New features and capabilities are incorporated with each version of Python. This has made Python the preferred choice of developers working in different application domains such as machine learning, GUI construction, API development, etc.

With the inclusion of support for asynchronous processing, using Python in building high-performance web apps has become increasingly prevalent. FastAPI is one of the fastest web application frameworks. It implements the ASGI (**Asynchronous Server Gateway Interface**) specification.

FastAPI is a relatively young framework. Yet it has become quite popular with the developer community. This book aims to help the reader get acquainted with its salient features. Experienced Python developers looking to leverage the flexibility of Python and the powerful features introduced in modern Python as well as computer science engineering students at graduate and postgraduate levels will also benefit immensely from the practical approach adapted in the book.

How This Book Is Arranged

This book comprises ten chapters.

Chapter 1: To begin with, Python's type hinting and its handling of asynchronous process are the two aspects introduced. FastAPI is built on top of **Starlette** and **Pydantic**. In this chapter, the reader is introduced to these two libraries.

INTRODUCTION

Chapter 2: FastAPI follows OpenAPI standards and integrates seamlessly with **Swagger UI**. In this chapter, the reader will learn how FastAPI is able to autogenerate the documentation for the API endpoints.

Chapter 3: This chapter deals with Pydantic's **BaseModel** and how it populates the body of an HTTP request.

Chapter 4: Although FastAPI is primarily a tool for API development, it can also be used to build web applications that render web pages and serve static assets. This chapter covers how to use **jinja2** templates and include static files.

Chapter 5: This chapter explains how the FastAPI application inserts cookies and headers in its response and how it is able to retrieve them.

Chapter 6: This chapter provides a comprehensive explanation of using SQL and NoSQL databases as the back end for a FastAPI application.

Chapter 7: This chapter is the beginning of the advanced part of this book. It introduces APIRouters with which bigger applications can be modularly constructed.

Chapter 8: Apart from REST, FastAPI supports **WebSocket** and **GraphQL** protocols. This chapter describes how to design apps that implement WebSocket and GraphQL.

Chapter 9: This chapter explains how the reader can secure the API with different provisions in FastAPI. The reader is also introduced to FastAPI's testing functionality.

Chapter 10: Deploying your API for public availability is very important. This chapter discusses different deployment alternatives.

As mentioned earlier, this book is intended to be a hands-on guide to learn FastAPI. Hence, it is replete with code listings and screenshots, which should help the reader to learn the concepts by executing the code as they read along. All the code snippets are thoroughly tested and are available in the repository: <https://github.com/Apress/Build-High-Performance-Web-Apps-with-FastAPI-by-Malhar-Lathkar>.

Developing the content of this book has been an extremely enjoyable process. Hopefully, it will prove to be equally enjoyable for the reader.

CHAPTER 1

Introduction to FastAPI

The recent surge in the popularity of Python as a programming language is mainly due to its libraries used in the field of data science applications. However, Python is also extensively used for web application development, thanks to the abundance of its web application frameworks.

FastAPI is the latest entrant in the long list of Python’s web application frameworks. However, it’s not just another framework as it presents some distinct advantages over the others. Considered to be one of the “fastest,” FastAPI leverages the capabilities of modern Python. In this chapter, we shall get acquainted with the important features on top of which the FastAPI library is built.

This chapter covers the following topics:

- Type hints
- Asynchronous processing
- REST architecture
- HTTP verbs
- FastAPI dependencies
- FastAPI installation

Type Hints

Python is a dynamically typed language. In contrast, the languages C/C++ and Java are statically typed, wherein the type of the variable must be declared before assigning a value to it. During the lifetime of a C/C++/Java program, a variable can hold the data of its declared type only. In Python, it is the other way round. The type of the variable is decided by the value assigned to it. It may change dynamically on each assignment. The interaction in the Python console in Listing 1-1 shows Python's dynamic typing.

Listing 1-1. Dynamic typing

```
>>> x=10
>>> #x is an int variable
>>> x=(1,2,3)
>>> type(x)
<class 'tuple'>
>>> x=[1,2,3]
>>> #x is now a list
>>> type(x)
<class 'list'>
>>> x="Hello World"
>>> #x now becomes a str variable
>>> type(x)
<class 'str'>
```

Although this dynamic typing feature makes programming easier, it also becomes prone to encountering runtime errors, as the Python interpreter doesn't enforce any type checking before executing. Take a look at the example in Listing 1-2.

Listing 1-2. Python function

```
#hint.py
def division(num, den):
    return num/den
```

Let us import this function and call it from the Python prompt as shown in Listing 1-3.

Listing 1-3. TypeError in the function

```
>>> from hint import division
>>> division(10,2)
5.0
>>> division(10,2.5)
4.0
>>> division("Python",2)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    division("Python",2)
  File "F:\python36\hint.py", line 3, in division
    return num/den
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The first two calls to the `division()` function are successful, but the `TypeError` exception is thrown for the third case, because the division operation of a numeric and a nonnumeric operand fails.

The default Python interpreter (Python shell) that comes with the standard installation of Python is rather rudimentary in nature. The more advanced Python runtimes such as IPython and Jupyter Notebook as well as many Python IDEs like VS Code, PyCharm (including IDLE – a basic IDE shipped with the standard library) are more intelligent, having useful features such as autocompletion, syntax highlighting, and type ahead help. Figure 1-1 shows the autocompletion feature of IPython.

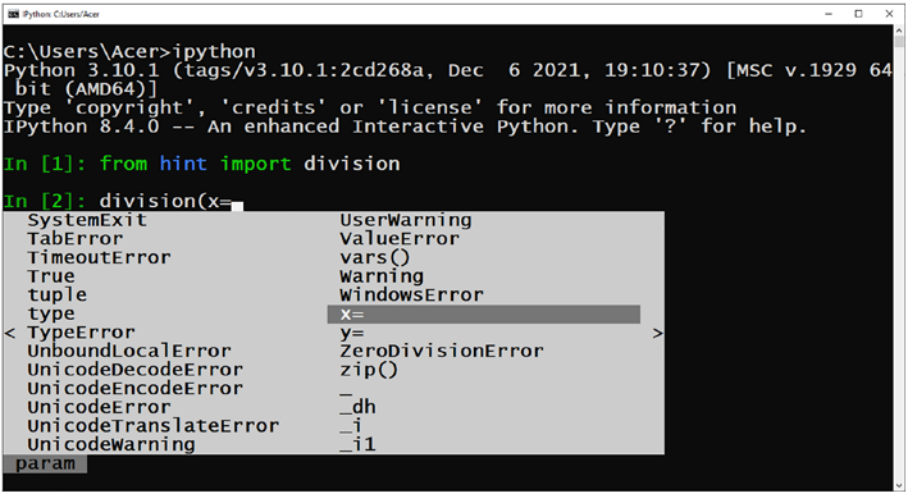


Figure 1-1. IPython shell

The type hinting feature was introduced in version 3.5 of Python. When a variable is used, its expected type can now be mentioned after the : (colon) symbol in front of it. The following definition of the `division()` function indicates the data types of its parameters:

```
def division(num:int, den:int):  
    return num/den
```

Although the Python interpreter still doesn't enforce type checking, the annotation of parameters with data types is picked by Python IDEs. The IDE lets the user know what types of values are to be passed as arguments to the function while calling.

Let us see how **VS Code** – a very popular IDE for program development not only in Python but in many other languages too – reacts to the type hints. In Figure 1-2, we find that the preceding function is defined and its return value is displayed.

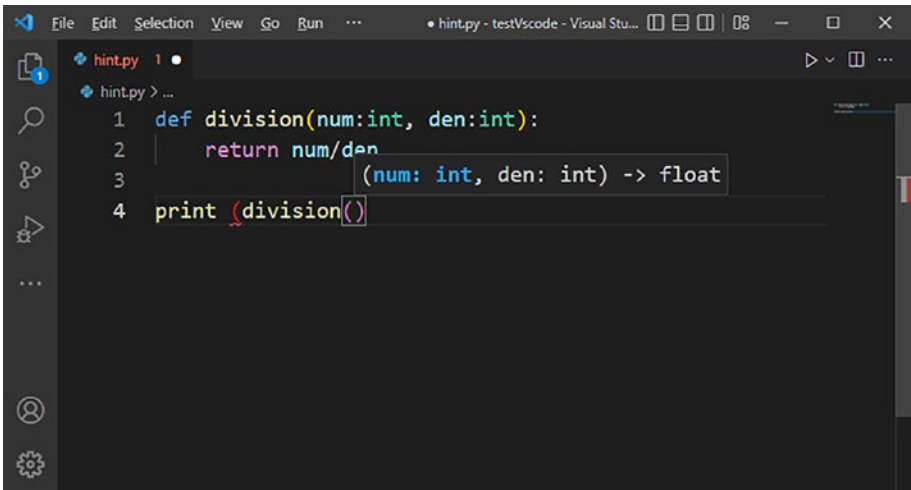


Figure 1-2. Type hints in the VS Code editor

As soon as the left parenthesis key is entered while calling the `division()` function, VS Code pops the signature of the function, indicating (hinting) to the user that the function needs two arguments of `int` type.

We can also provide the type hint for the return value of the function. Put an `(->)` arrow symbol after the closing parenthesis of the definition and mention the desired type after it before starting the function's code block. Let us add `float` as the type hint for the return type in the definition of the `division()` function in the code snippet in Listing 1-4.

Listing 1-4. Function with type hints

```
def division(num:int, den:int) -> float:
    return num/den
```

All the standard types, those defined in imported modules, and user-defined types can be used as hints (Listing 1-5). Type hints can be used for global variables, function and method parameters, as well as local variables inside them.

Listing 1-5. hintexample.py

```
#hintexample.py
arg1: int = int(input("Enter a number.."))
arg2: int = int(input("Enter a number.."))

def division(num:int, den:int) -> float:
    result:float = num/den
    return result

print (division(arg1, arg2))
```

As mentioned earlier, using type hints doesn't cause the interpreter to enforce type checking at runtime. We can, however, use a static type checker utility like **mypy** to detect type mismatch errors before running.

The typing Module

This module is a new addition to Python's standard library. It enhances the type hinting feature by introducing a special collection of data types – List, Tuple, and Dict (note that first uppercase character of each type), each corresponding to Python's built-in types list, tuple, and dict. The built-in collection objects can contain items of any Python type. On the other hand, we can specify the type for the items in a collection in the form of a hint. The difference can be seen in the statements in Listing 1-6.

Listing 1-6. Difference between list and List

```
>>> l1:list = ["Python", 10, 1.5E2, True]
>>> import typing
>>> l2:typing.List[float]= [100, 25.50, 2.2E-2]
```

Here, `l1` is a standard list and is a collection of items of different types. On the other hand, `l2` is of `typing.List` type expected to hold items of float type only. Needless to say, this hint is disregarded by the Python runtime if an object of nonfloat type is appended without error, as in Listing 1-7.

Listing 1-7. Append operation of `typing.List`

```
>>> l2.append("hello")
>>> l2
[100, 25.5, 0.022, 'hello']
```

The `typing` module also defines `Union`, `Any`, and `Optional` types. The `Union` type should be used as a hint to provide a list of possible data types for an object. Consider the case in Listing 1-8 where each item in this list is expected to be of either `int` or `str` type.

Listing 1-8. Union type in the `typing` module

```
>>> from typing import Tuple, Union
>>> l3:Tuple[Union[int, str]]= [5, "Python", 100]
```

Use `typing.Any` to indicate that there's no constraint on the type of the variable. The `Optional` type means that the object can be either of the specified type or it can be `None` as in Listing 1-9.

Listing 1-9. Optional type in the `typing` module

```
>>> from typing import Optional
>>> obj=Optional[str]
>>> obj="Hello World"
>>> obj=None
```

You can of course assign any other value; the static type checker linters will detect the mismatch.

We have discussed the type hint mechanism in brief here, mainly because FastAPI uses this feature very extensively. You will find this type hint syntax in the declaration of path and query parameters in the definition of view functions. It is also used in **Pydantic** model declarations. An important feature of FastAPI – the autogeneration of API documentation – also depends on the type hinting feature of Python.

Asynchronous Processing

Python supports concurrent processing by using a multithreading approach ever since its earlier versions. The asynchronous processing support has been added to Python from the Python 3.5 version onward. In a multithreaded application, many individual threads of operation are there in the program, and the CPU coordinates their concurrent execution. On the other hand, in the asynchronous approach, only a single thread runs, but it has the ability to switch between functions.

Whenever an asynchronous function reaches an event or condition, it voluntarily yields to another function. By the time the result from the other function is obtained, the original function can attend some other operations. In this way, more than one process in an application can run concurrently, without intervention from the operating system. Moreover, as there's a single running thread, it doesn't involve heavy processor resources. Asynchronous processing is also sometimes (and appropriately) called cooperative multitasking.

The asyncio Module

Python's asynchronous capabilities come from the functionality defined in the built-in `asyncio` module and the two newly added keywords – `async` and `await`. The `asyncio` module serves as the foundation for Python's ASGI (**Asynchronous Server Gateway Interface**) compatible web application frameworks like FastAPI.

In an asynchronous application, the processing logic is structured in coroutines instead of the traditional functions. A coroutine is similar to a function, but can yield to another coroutine and awaits its completion. A **coroutine** is a function with an `async` keyword as a prefix in the definition.

The coroutine is not called like a normal Python function. If we try to call, it just returns the coroutine object. To call a coroutine, use the `run()` function in the `asyncio` module.

The code snippet in Listing 1-10 shows the difference between a function and a coroutine.

Listing 1-10. Difference between a function and a coroutine

```
>>> #This is a normal function
>>> def hello():
    print ("Hello World")
>>> #This is a coroutine
>>> async def sayhello():
    print ("Hello Python")

>>> hello()
Hello World
>>> sayhello()
<coroutine object sayhello at 0x0000015CC0295DB0>

>>> import asyncio
>>> asyncio.run(sayhello())
Hello Python
```

Cooperative multitasking comes into play when one coroutine “awaits” another. When the `await` keyword is encountered in the path of execution of a subroutine, its operation is suspended till the other coroutine is completed.

The example in Listing 1-11 demonstrates how the `async/await` mechanism works. The coroutine named `main()` has a `for` loop. For each iteration, it waits for another coroutine called `myfunction()`. The second

coroutine prints the iteration number, pauses for two seconds, and goes back to `main()`. The outer coroutine then prints the number and goes for the next iteration.

Listing 1-11. `coroutines.py`

```
#coroutines.py
import asyncio
import time

async def main():
    for i in range(1,6):
        await myfunction(i)
        print ('In main', i)

async def myfunction(i):
    print ('In myfunction', i)
    time.sleep(2)

asyncio.run(main())
```

Go ahead and execute this Python script. The output shows how the two coroutines take turns concurrently:

```
In myfunction 1
In main 1
In myfunction 2
In main 2
In myfunction 3
In main 3
In myfunction 4
In main 4
In myfunction 5
In main 5
```

ASGI

Many of the well-known web application frameworks of Python (like **Django** and **Flask**) were developed before the introduction of asynchronous capabilities. These frameworks implement WSGI (**Web Server Gateway Interface**) specifications. The request-response cycle between the client and the server is synchronous in nature and hence not particularly efficient.

To leverage the functionality of the `asyncio` module, new specifications for web servers, frameworks, and applications have been devised in the form of ASGI, which stands for **Asynchronous Server Gateway Interface**. An ASGI callable object is in fact a coroutine having three parameters:

- **scope**: A dict containing details of a specific connection provided by the server.
- **send**: An asynchronous callable, by which event messages can be sent to the client.
- **receive**: Another asynchronous callable. The application can receive event messages from the client.

Listing 1-12 shows a very simple ASGI callable coroutine. It calls the `send` coroutine as an awaitable object to insert the HTTP headers and the body in the client's response. These calls are nonblocking in nature so that the server can engage with many clients concurrently.

Listing 1-12. ASGI application coroutine

```
async def app(scope, receive, send):
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            [b'content-type', b'text/plain'],
```

```

    ],
})

await send({
    'type': 'http.response.body',
    'body': b'Hello, world!',
})

```

In order to serve this application, we need a web server having asynchronous capabilities. Python’s `wsgiref` module does have a built-in development server to launch a WSGI application. However, a corresponding ASGI server is not shipped with Python’s standard library. Hence, we need to use a third-party ASGI server called **Uvicorn**. With its help, we can fire our ASGI application. Save the code in Listing 1-13 as `main.py`.

Listing 1-13. `main.py`

```

# main.py
import uvicorn

async def app(scope, receive, send):
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            [b'content-type', b'text/plain'],
        ],
    })

    await send({
        'type': 'http.response.body',
        'body': b'Hello, world!',
    })

```

```
if __name__ == "__main__":
    uvicorn.run("main:app", port=5000, log_level="info")
```

Run this program from the command line (make sure that the Uvicorn package is installed before running). The Uvicorn server starts serving the applications and is waiting for incoming requests from a client at port 5000 of the localhost. Launch your favorite browser and visit the `http://localhost:5000` URL. The browser window shows (Figure 1-3) the Hello World message as the response.



Figure 1-3. ASGI application

The good thing about Python’s application frameworks is that they make their own application function available, and one needn’t develop it manually as in the preceding case. FastAPI is an ASGI-compliant framework. An object of the FastAPI class itself is the ASGI callable and hence is used as a parameter to the `uvicorn.run()` function call in the preceding code.

About REST Architecture

The FastAPI library, which we are going to learn about in detail in this book, according to its official documentation, is a “*modern, fast web framework for building APIs*.” It is therefore imperative that we understand more about APIs and in particular the REST architecture.

What Is an API?

The term **API** is very frequently used within the web developer community. It is an acronym for Application Programming Interface. The word **interface** generally refers to a common meeting ground between two isolated and independent environments. The interaction between them takes place as per a predefined set of rules and protocols. To give a simplistic analogy, the waiter in a restaurant acts as an interface between the customer and the kitchen, accepting the order for the items in the menu, passing on the order to the kitchen staff for preparing the dish, and in turn serving it to the customer. Figure 1-4 depicts the analogy between the roles of a waiter and an API.

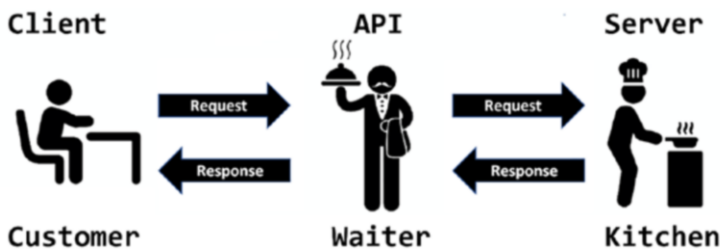


Figure 1-4. *Analogy between a waiter and an API*

In a computer system, electromechanical peripheral devices, such as a mouse, printer, etc., communicate with the CPU via interfaces (serial, parallel, USB, etc.). A programming interface on the other hand is an interface between two software applications. A stand-alone application is designed to accept user input in a predefined format, perform database read/write operations if needed, process the data at the back end, and present the result on available output devices. But what if this application is to be made available to other users not having access to the machine on which it is installed?

Consider a fast food company (such as McDonald’s) which has developed a web application for its customer to book orders. A customer

can of course visit the company's website, go through the registration and authentication process, and order the food of their choice. Now you have a food delivery service (such as Grubhub in the United States, Swiggy in India) that wants to enable its clients to use the order booking system. Obviously, an access by an unauthorized user will be denied. (Or else, the situation will be akin to each customer in the restaurant directly going to the kitchen and ordering the chef to prepare a certain dish that is not even on the menu!)

This is where the API comes into play. The company will deploy a system wherein the delivery agency website will be authorized and asked to place the order in a specified format. The request received will then be processed appropriately by the company's application, and the result will also be sent to the agency in a predefined format for further consumption. This mechanism that facilitates communication between two different applications (here, the company's order processing application and the delivery agency's application) by following certain streamlined formats and protocols is what makes an API. Please refer to Figure 1-5 for better understanding.

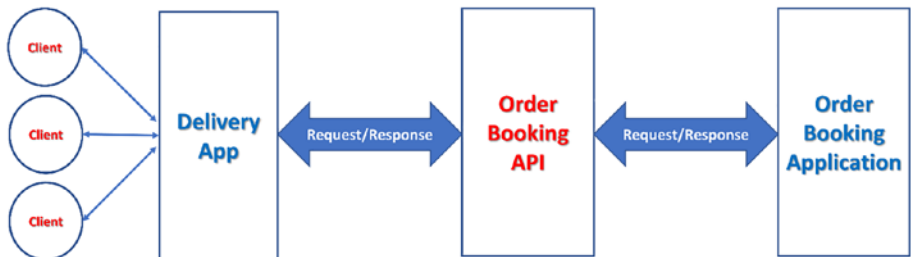


Figure 1-5. Example of an API

Registration and authentication of a user with the APIs provided by social platforms are being popularly employed by different web apps. Instead of registering with an app directly with one's email/password or

mobile number/OTP, it is convenient to use the OAuth API of social media services like Facebook, Twitter, etc. Figure 1-6 shows the login page of the New York Times website.

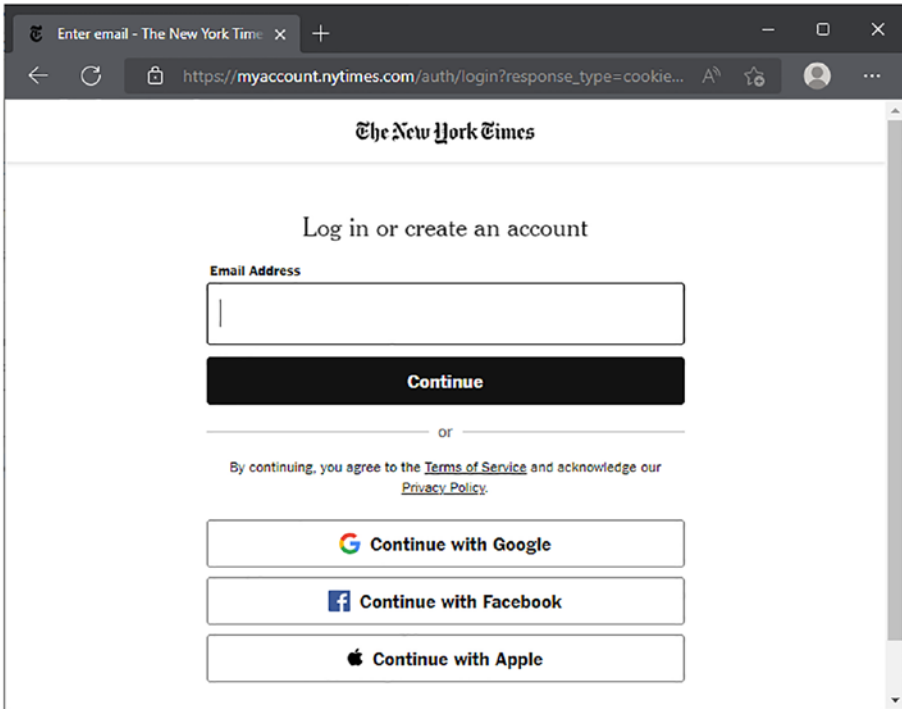


Figure 1-6. Login screen of the New York Times portal

A web API is a web application that exposes its resources to be accessed by other web/mobile applications through the Internet. It defines one or more endpoints which the client apps can visit to perform read/write operations on the host's resources. Over the years, the REST architecture has become the de facto standard for building APIs. Although there are other approaches available for developing an API (such as RPC, stands for Remote Procedure Call, and SOAP, stands for Simple Object Access Protocol), REST is very flexible and thus gives a lot of freedom for developers.

REST

REST (short for **REpresentational State Transfer**) is a software architectural style that defines how a web application should behave. The term REST was first introduced by **Roy Fielding** in the year 2000. A web API that follows this architectural style is often called the RESTful API.

Unlike **RPC** or **SOAP**, REST is not a protocol. Moreover, REST is a resource-based architecture rather than action based as is the case in RPC or SOAP. Everything on the REST server is a resource, may it be a file, an image, or a row in a table of a database. The REST API provides a controlled access to its resources so that the client can retrieve and modify them.

REST Constraints

A web API based on the REST architecture must follow certain design principles, also called constraints, laid down by Roy Fielding in his research paper. The following are the six REST constraints.

Uniform interface: This constraint essentially means that the request for a certain resource on the server from any client must be the same. In other words, a resource on the server must have a Uniform Resource Identifier (URI), and the request for a resource should be complete in the sense it should contain all the information required for retrieving and processing it.

Statelessness: When the server receives any request for a certain resource from a client, it should be processed entirely in isolation without any context of any previous transaction. Any change in the state of the concerned resource is communicated back to the client. REST is implemented over HTTP, and the HTTP server doesn't retain the session information, thereby increasing the performance by removing server load.

Client-server: The client-server model is also the backbone of HTTP. It ensures that there is a separation of concerns, leading to the evolution of server and client applications independent of each other. As long as the

interface between them is not altered, servers and clients may also be replaced and developed independently. In fact, the client is expected to know only the URIs of the resources on the client and nothing else.

Cacheability: The term caching refers to storing the transient data in a high-speed buffer so that subsequent access to it is faster. By marking the API response as cacheable, it improves the performance of the client, as well as the scalability of the server.

Layered system: Just as the server and client components are isolated from each other (thanks to the client-server constraint of REST), the server functionality itself can be further composed into multiple hierarchical layers, each independent of the other. A certain layer designed to perform a specific task can interact with only the immediate layer and none other. This approach improves system scalability and enables the load balancing of services across multiple networks and processors.

Code on demand: On most occasions, the server's response is in either the HTML or XML representation of the resource. However, according to this constraint (which by the way is the only optional feature among the six), the response may return some front-end executable component such as JavaScript. The client can download the same. Though this feature improves the extensibility, it can also be a potential security issue, hence rarely implemented.

In a nutshell, implementing the preceding principles (constraints) has the following advantages:

- Scalability
- Simplicity
- Modifiability
- Reliability
- Portability
- Visibility

HTTP Verbs

The principle of uniform interface says that the request message must be self-sufficient and that it should contain everything that is required to process the request. It should include the URI of the resource, the action to be taken on it, and some additional data if required for the action to be completed. The action part in the request is represented by HTTP verbs or methods. The most used HTTP methods are **POST**, **GET**, **PUT**, and **DELETE**. These methods correspond to CREATE, READ, UPDATE, and DELETE operations on the server's resource. These operations are popularly known by the abbreviated form **CRUD**.

In addition to the abovementioned methods, the HTTP request type can be of few other types (those being **PATCH**, **HEAD**, **OPTIONS**, etc.). However, they are less prevalent, especially in the context of REST architecture.

So, we can draw the inference that the client sends an HTTP request of either POST, GET, PUT, or DELETE type to perform a CRUD operation on a certain resource residing with the server. This request is intercepted by the REST API designed as per the constraints explained earlier and forwarded to the server for processing. In turn, the response is returned to the client for its further consumption.

POST Method

The POST verb in the HTTP request indicates that a new resource is intended to be created on the server. It corresponds to the CREATE operation in the CRUD term. Since the creation of a new resource needs certain data, it is included in the request as a data header. If the request is successful, the HTTP response returns a Location header with a link to the newly created resource with the 201 HTTP status. In case the operation is not successful, the status code is either 200 (OK) or 204 (No Content).

Since invoking two identical POST requests will result in two different resources containing the same information, it is not considered as **idempotent**.

Examples of a POST request:

```
HTTP POST http://example.com/users
```

```
HTTP POST http://example.com/users/123
```

GET Method

The READ part in the CRUD operation is carried out by sending the HTTP request with a GET method. The purpose of the GET operation is to retrieve an existing resource on the server and return its XML/JSON representation as the response. Success inserts the 200 (OK) status code in the response, whereas its value is 404 (Not Found) or 400 (Bad Request) in case of failure.

The GET operation is risk-free in the sense that it only retrieves the resource and doesn't modify it. Furthermore, it is considered to be an idempotent operation, as two identical GET requests return identical responses.

Examples of a GET request:

```
HTTP GET http://example.com/users
```

```
HTTP GET http://example.com/users/123
```

PUT Method

The PUT method is used mainly to update an existing resource (corresponding to the UPDATE part in CRUD). Like the POST operation, the data required for update should be included in the request. Success returns a 200 (OK) status code, and failure returns a 404 (Not Found) status code.

In some cases, the PUT operation may create a new resource, depending on how the business logic of the API is written. In such a case, the response contains a 201 status code. The response of the PUT method generally doesn't contain the representation of the newly created resource.

The difference between the POST and PUT is that POST requests are made on resource collections, whereas PUT requests are made on a single resource.

Examples of a PUT request:

```
HTTP PUT http://example.com/users/123
```

```
HTTP PUT http://example.com/users/123/name/Jeevan
```

DELETE Method

As the name suggests, the DELETE method is used to delete one or more resources on the server. On successful execution, an HTTP response code 200 (OK) is sent. It may also be 202 (Accepted) if the action has been queued or 204 (No Content) if the action has been performed but the response does not include an entity.

DELETE operations are idempotent, like the GET method. If a resource is deleted, it's removed from the collection of resources. However, calling DELETE on a resource a second time will return a 404 (Not Found) since it was already removed.

Examples of a DELETE request:

```
HTTP DELETE http://example.com/users/123
```

```
HTTP DELETE http://example.com/users/123/name/Jeevan
```

FastAPI Dependencies

FastAPI is an asynchronous web framework that incorporates Python's type annotation feature. It is built on top of **Starlette** – Python's ASGI toolkit. FastAPI also uses **Pydantic** as an important building block for validating the data models. An ASGI application should be hosted on an asynchronous server; the fastest one around is **Uvicorn**.

FastAPI also integrates seamlessly with **OpenAPI** (earlier known as Swagger) for API creation. It also generates data model documentation automatically, based on JSON Schema.

Figure 1-7 shows the relationship between various dependency libraries.

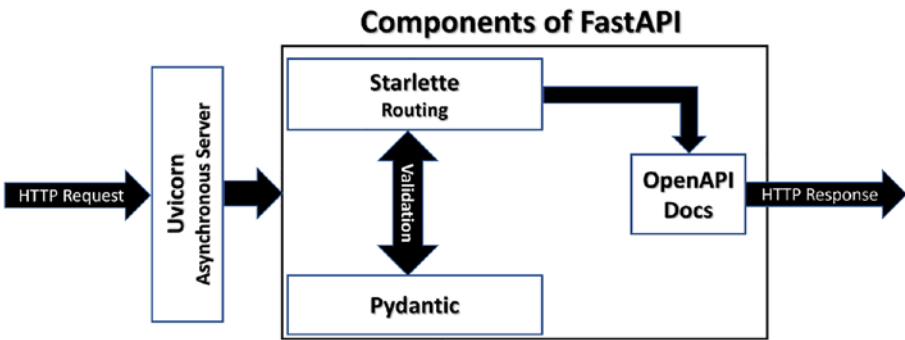


Figure 1-7. FastAPI dependencies

Starlette

Starlette is a lightweight ASGI-compliant web framework, ideal for leveraging the advantage of `asyncio` in building high-performance APIs and web apps. Its simple and intuitive design makes it easily extensible. Some of the important features of Starlette include the following.

WebSocket support: WebSocket is the extension of the HTTP protocol to provide bidirectional, full-duplex communication between the client and the server. WebSockets are used for building real-time applications.

Event handlers: Starlette supports intercepting and processing startup and shutdown events in a web application. They can be effectively used to handle certain initialization and mopping up tasks such as establishing and closing the database connection as the application starts and closes.

The Starlette framework has all the essential functionality required to build a web app. It includes routing, templating, serving static files, cookie and session support, etc. FastAPI is built on top of Starlette. In addition, FastAPI incorporates additional features such as the dependency injection system, security utilities, OpenAPI schema generation, etc.

Pydantic

Pydantic is another library that FastAPI uses as an important pillar. It is an excellent Python library for data validation and parsing. FastAPI uses it to describe the data part of a web application. It uses Python's typing annotation feature, enforcing type hints at runtime. Pydantic maps the data to a Python class.

Using Pydantic data models makes it very easy to interact with relational database ORMs (like **SQLAlchemy**) and ODMs (like **Motor** for MongoDB); it takes care of the data validation aspect very efficiently.

Pydantic is capable of validating Python's built-in types, user-defined types, the types defined in the typing module (such as List and Dict), as well as complex data types involving Pydantic's recursive models.

We shall discuss Pydantic models in more detail in one of the subsequent chapters.

Uvicorn

As mentioned earlier, Uvicorn is an asynchronous web server with high performance. It has been designed as per the ASGI specifications defined in the `asgiref` package. This package has an `asgiref.server` class which is used as the base.

To enable WebSocket support, a standard version of Uvicorn needs to be installed. It brings in Cython-based dependencies – `uvloop` and `httptools`. The `uvloop` library provides a more efficient replacement for the event loop defined in the `asyncio` module. The `httptools` on the other hand is used to handle the HTTP protocol.

By default, the Uvicorn server binds a TCP socket to the **localhost** (127.0.0.1) and listens on port 8000; both can be customized at runtime if so required.

Uvicorn supports the **HTTP/1.1** version. For the HTTP/2 version, you need to employ **Daphne** or **Hypercorn** web server software.

Installation of FastAPI

Now that we have understood the important prerequisites, it is now time to introduce the FastAPI web application framework. FastAPI is a web application framework based on Python’s modern features such as type hints and support for asynchronous processing. The async feature makes it extremely “fast” as compared with the other Python web frameworks.

FastAPI was developed by **Sebastian Ramirez** in December 2018. FastAPI 0.79.0 is the currently available version (released in July 2022). In spite of being very young, it has very quickly climbed up on the popularity charts and is one of the most loved web frameworks.

So, let’s go ahead and install FastAPI (preferably in a virtual environment). It’s easy. Just use the **PIP** utility to get it from the PyPI repository. Ensure that you are using Python’s 3.6 version or later:

```
pip3 install fastapi
```

Since FastAPI is built on top of two important libraries, Starlette and Pydantic, they are also installed, along with some others.

You also need to install the Uvicorn package to serve the FastAPI app:

```
pip3 install uvicorn[standard]
```

The “standard” option installs the Cython-based dependencies, uvloop, httptools, and websockets. If you don’t intend to use WebSockets, this option may be omitted. Certain additional supporting libraries are also installed.

To get the list of packages installed, use the **freeze** subcommand of the PIP utility (Listing 1-14).

Listing 1-14. Packages installed

pip3 freeze

```
anyio==3.6.1
click==8.1.3
colorama==0.4.5
fastapi==0.79.0
h11==0.13.0
httptools==0.4.0
idna==3.3
pydantic==1.9.1
python-dotenv==0.20.0
PyYAML==6.0
sniffio==1.2.0
starlette==0.19.1
typing_extensions==4.3.0
uvicorn==0.18.2
watchfiles==0.16.0
websockets==10.3
```

- anyio is an asynchronous networking and concurrency library that implements a structured concurrency on top of asyncio.
- click stands for **Command Line Interface Creation Kit**. This package is required if you intend to use **typer** for building a CLI instead of a web API.
- colorama is a cross-platform library to render colored text in the Python terminal.

- The `h11` package is used internally by the Uvicorn server to implement the HTTP/1.1 protocol.
- The `typing-extensions` module acts as a backport for Python 3.6–based applications. It may not be used if you are using newer versions of Python (version 3.7+).
- The `watchfiles` package is a simple, modern, and high-performance file watching and code reload in Python. To autoreload the application code when it is already running on the server, the `watchfiles` package is needed.
- The `sniffio` package detects which async library is being used by your code.
- Using `python-dotenv` is a convenient way to load the environment variables from a `.env` file.

Let us conclude this chapter by writing a simple “Hello World” app using FastAPI. Save the code in Listing 1-15 as `main.py` in the newly created FastAPI environment.

Listing 1-15. `main.py` (Hello World with FastAPI)

```
#main.py
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
async def index():
    return {"message": "Hello World"}
```

Without bothering to understand how this code works, execute the following command in the terminal window of your OS:

```
uvicorn main:app --reload
```

The terminal shows the following log, effectively telling that the application is being served at port 8000 of the localhost:

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press
          CTRL+C to quit)
INFO:      Started reloader process [28720] using WatchFiles
INFO:      Started server process [28722]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Open a browser window and enter `http://localhost:8000/` in its address bar to obtain the output as shown in Figure 1-8.

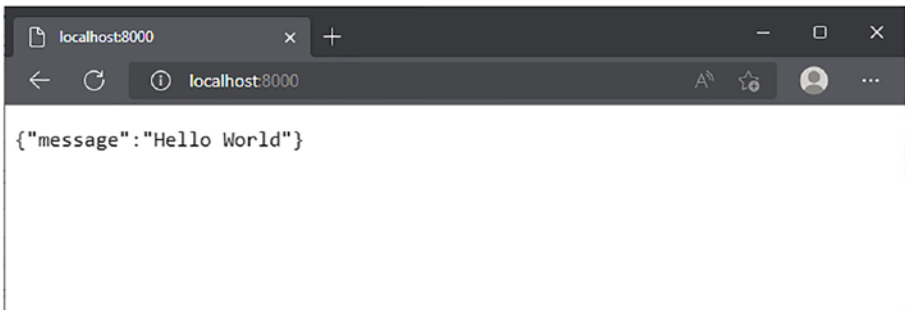


Figure 1-8. *Hello World with FastAPI*

Your FastAPI app is up and running!

Summary

This chapter set the ball rolling for our journey of learning how to build APIs with the FastAPI framework. In this chapter, we learned about the modern concepts of Python (typing and `asyncio` modules), which are very crucial in using FastAPI. We also took a quick revision of the principles of REST architecture. After introducing the dependency libraries of FastAPI – namely, `Starlette`, `Pydantic`, and `Uvicorn` – we actually installed it and successfully executed a Hello World application.

In the next chapter, we shall discuss in detail the elements of a FastAPI application such as the path operations, decorators, and views. You will also understand how interactive API docs are generated and how to pass path and query parameters to the view function.

CHAPTER 2

Getting Started with FastAPI

The previous chapter laid the groundwork for exploring the powerful features of the FastAPI web framework. We now know enough about the type hinting and asynchronous processing mechanism that is extensively implemented in FastAPI. It is primarily a tool for developing web APIs. Hence, a brief discussion on the principles of REST architecture was also given in the previous chapter.

This chapter helps you take the first steps toward building APIs. We shall also learn about the API docs and parameters.

We shall discuss the following topics:

- Hello World
- Interactive API docs
- Path parameters
- Query parameters
- Validation of parameters

Hello World

Toward the end of the previous chapter, you saw a small FastAPI script that renders a Hello World message in the browser. Let us revisit that script and understand how it works in detail.

Conventionally, the first program written whenever you learn a new programming language or a framework is the one that displays a “Hello World” message. The objective is to verify that the respective environment is correctly installed. It also helps in understanding the basic building blocks of the application.

Create an Application Object

First of all, we need an ASGI callable to be run by the server. The FastAPI library itself provides the callable in the form of an object of the FastAPI class. So, instantiate the object with the following statements (first two lines in the main.py in the previous chapter; please refer to Listing 1-15):

```
from fastapi import FastAPI
app = FastAPI()
```

This app object is the main point of interaction between the ASGI server and the client, as it is responsible for routing all the incoming requests to the appropriate handlers and providing appropriate responses.

Path Operation Decorator

In a classical web application, the URL requested by the client browser refers to a server-side script stored as a file on the web server. For instance, look at the URL <http://mysite.com/hello.php>, where the server runs the script file and renders its output as the response to the client. Sometimes, one or more parameters may be appended to the URL as a query string and meant to be processed by the script – such as <http://mysite.com/hello.php?name=Rahul&marks=65>.

Modern web application frameworks use a route-based approach to form the URL rather than the file-based URL. It happens to be more convenient and easier for the user. The application object maps a predefined URL pattern with a certain function, whose return value in turn becomes the server's response.

The route-based version of the URL shown earlier becomes <http://mysite.com/Rahul/65>. The URL has three distinct parts: the protocol (such as `http://` or `https://`) followed by the IP address or hostname. The remaining part of the URL after the first / after the hostname is called the path or endpoint.

In the previous chapter, to obtain the Hello World message as the result, we used `http://localhost:8000/` as the URL. Since there's nothing after the first /, it becomes the path or the endpoint.

The server also needs to know the HTTP method (either GET, POST, PUT, or DELETE) used by the client to send the request. In FastAPI, as per the **OpenAPI** standards, these methods are called operations. So, the GET operation retrieves a resource, the POST operation creates a new resource, and the PUT and DELETE operations modify and delete a resource, respectively.

The FastAPI class defines path operation methods corresponding to HTTP operations mentioned earlier. They are `@app.get()`, `@app.post()`, `@app.put()`, and `@app.delete()`. These methods need a mandatory path parameter – as in `@app.get("/")`. The `@` symbol is prefixed to indicate that they are decorators.

A function in Python can accept another function as an argument, just as it can have other data types – `int`, `str`, `list`, or an object of another class. Similarly, inside the definition of one function, another function's definition can be wrapped. Moreover, a function's return value can also be a function.

A decorator receives another function as its parameter and returns the same by making certain modifications in its behavior.

Path Operation Function

The next three lines in our `main.py` code are shown in Listing 2-1.

Listing 2-1. Path operation function

```
@app.get("/")
async def index():
    return {"message": "Hello World"}
```

What does the preceding code segment do? Whenever the application object finds that the client has requested the “/” path with a GET request, the `index()` function defined just below should be called. In other words, the URL path “/” is mapped with the `index()` function, called the path operation function. It usually returns a dict object. Its JSON form is returned as the response to the client.

The Hello World application code is reproduced here for convenience (Listing 2-2).

Listing 2-2. Hello World

```
#main.py
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
async def index():
    return {"message": "Hello World"}
```

Start Uvicorn

As mentioned before, throughout this book, we are going to use the Uvicorn server to run a FastAPI application. It is an ASGI implementation for Python. You can start the Uvicorn server either by using the command-line interface of the Uvicorn library or programmatically by calling its `run()` function.

To start the server from the command line, enter the following statement in the command terminal of your OS:

uvicorn main:app --reload

By default, the Uvicorn server uses localhost (equivalent to the IP address 127.0.0.1) and listens for the incoming requests at port number 8000. Both parameters can be changed if required.

To invoke the server programmatically, import uvicorn in the code and call its `run()` function with the application object as a parameter. The code in Listing 2-3 calls the `run()` function.

Listing 2-3. Run Uvicorn programmatically

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def index():
    return {"message": "Hello World"}

if __name__ == "__main__":
    uvicorn.run("main:app", host="127.0.0.1", port=8000,
        reload=True)
```

You now have to run the preceding program in the command terminal as follows:

python main.py

In either case, the log in Listing 2-4 indicates that the application is running at port 8000 of the localhost.

Listing 2-4. Uvicorn console log

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press
          CTRL+C to quit)
INFO:      Started reloader process [28720] using WatchFiles
INFO:      Started server process [28722]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Visit the URL `http://localhost:8000/` using a browser. A GET request for the “/” path is sent to the server. The `index()` function mapped to this URL endpoint is executed. The Hello World message as its return value is sent back as the response (Figure 2-1).

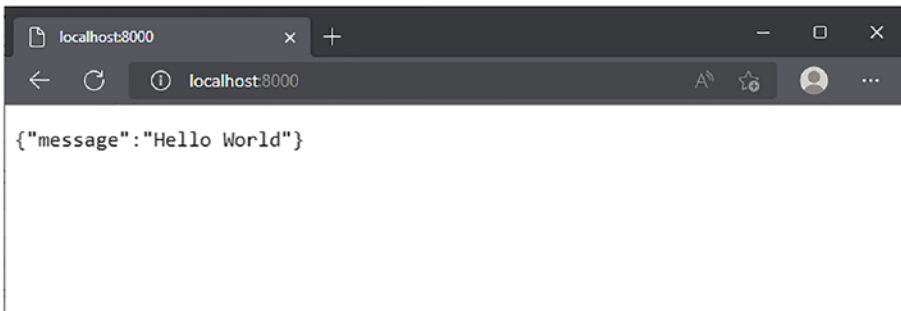


Figure 2-1. Hello World message in a browser

Externally Visible Server

As the host parameter is set to localhost (which is also its default value), the application is accessible only from the same machine on which it is running. To make it available for other devices, we need to take the following steps:

Either set the host parameter to 0.0.0.0 in the command line:

```
uvicorn main:app --host 0.0.0.0 -reload
```

or set the host parameter to this value in the call to the `run()` function in the `main.py` code (Listing 2-5).

Listing 2-5. Externally visible server

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def index():
    return {"message": "Hello World"}

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000,
        reload=True)
```

The application is now available to any other device on the same network. However, to access it, first find out the IP address of the machine on which the application is running. On Windows OS, it can be found by running the **ipconfig** command in the terminal.

Wireless LAN adapter Wi-Fi:

```
Connection-specific DNS Suffix . : ib-wrb304n.setup.in
Link-local IPv6 Address . . . . . : fe80::dd10:6074:1eb3:a43a%77
IPv4 Address. . . . . : 192.168.1.211
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.1
```

Open the browser on any other device on the same network, and enter the IP address shown in bold letters earlier (keep the port as 8000 as it hasn't been changed). The URL to be used is `http://192.168.1.211:8000`. The browser shows the same Hello World message as earlier.

Interactive API Docs

One of the standout features of FastAPI is its ability to automatically generate interactive API documentation. The Swagger UI tool integrates seamlessly with FastAPI. It provides a user-friendly web interface to visualize the documentation and exploration of the endpoints.

FastAPI's design follows the OpenAPI Specification (**OAS**) for API creation and declaration of path operations, parameters, etc. It also autogenerates the documentation of data models with JSON Schema. JSON Schema defines a JSON-based media type called “**application/schema+json**”. It is a format for describing the structure of JSON data. It specifies what a JSON document should be like, how to extract information from it, and how to interact with it.

Swagger is a suite of API development utilities, of which the **Swagger UI** is a part. It is a REST API development tool with a web interface. The Swagger specification is now a part of the Linux Foundation and has been renamed as OpenAPI. There are a number of such OpenAPI-compliant utilities. By default, FastAPI includes support for Swagger UI and **Redoc** (Figure 2-2).



Figure 2-2. API documentation tools

Swagger UI

Swagger UI's web interface is built with the help of HTML, JavaScript, and CSS assets to autogenerate an interactive documentation based on the API code. Here, we shall be using it along with the REST API written with FastAPI.

To understand how Swagger UI documentation works, let us first add one more path operation in our Hello World example. Update the `main.py` to the code shown in Listing 2-6.

Listing 2-6. Path operation with parameters

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
async def index():
    return {"message": "Hello World"}

@app.get("/{name}/{id}")
async def user(name:str, id:int):
    return {"name":name, "id":id}
```

The string path parameter of the second path operation decorator `@app.get()` has two identifiers enclosed in curly brackets. These are called path parameters. They are passed on to the operation function below it. We shall learn more about the path parameters in the next section. For now, go to the URL `http://localhost:8000/docs` after starting the Uvicorn server (use the same command as before).

This starts the Swagger UI tool and generates the documentation for the path operations from the application code. Figure 2-3 displays the operation, its path, and the mapped function for each decorator.

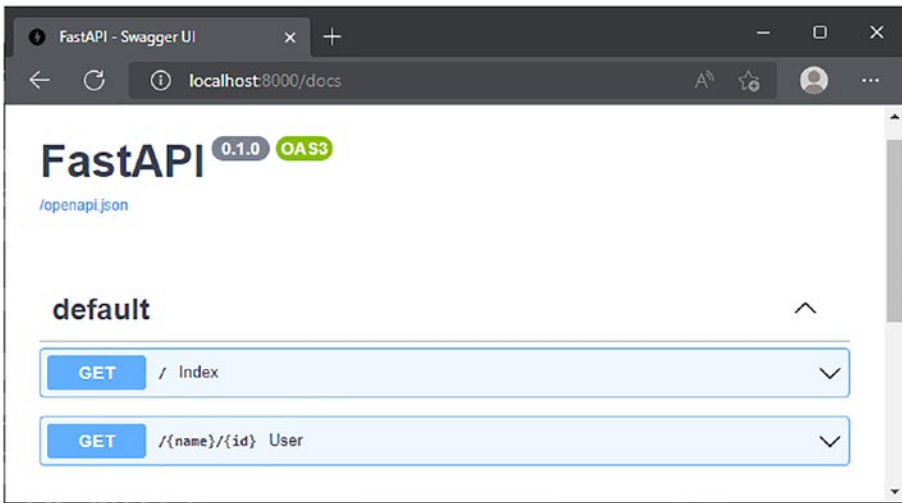


Figure 2-3. *Swagger UI*

Out of the two, expand the first operation corresponding to the `index()` function by clicking the arrow on the right-hand side. It reveals a **Try it out** button (refer to Figure 2-4) and shows if there are any parameters defined for the function. Since there aren't any, the Parameters list is empty.

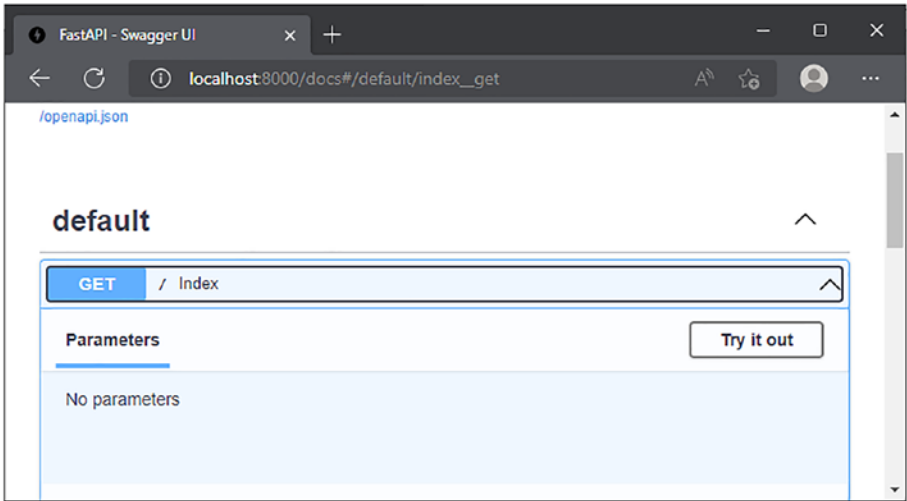


Figure 2-4. *Operation function without parameters*

Click the **Try it out** button. If there are any parameters, you have to provide their values (in the next section, we shall see how the path parameters are passed). A button with the caption **Execute** appears (Figure 2-5) in the window.

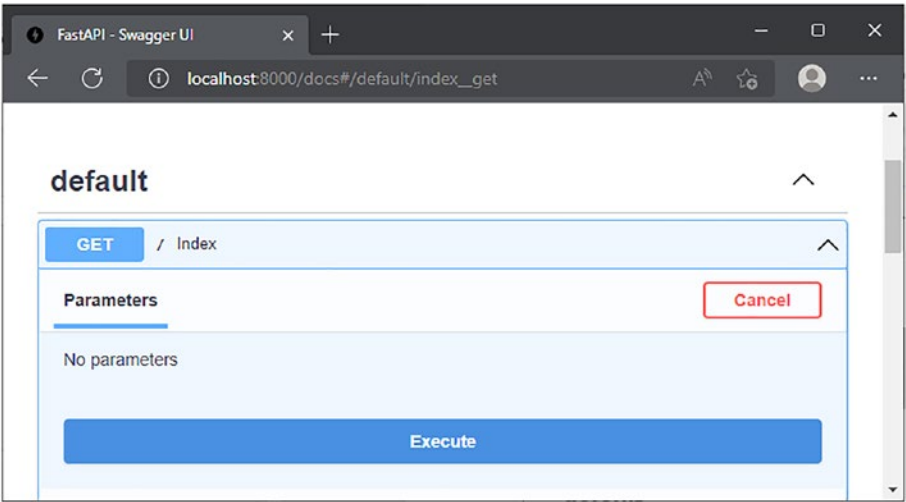


Figure 2-5. Test the function

As you click the **Execute** button, Swagger UI generates the documentation of request and response objects, as shown in Figure 2-6.



Figure 2-6. Server response

It shows the **Curl** representation of the operation. Here, no parameters have been passed. Hence, none are seen in the request URL. If indeed the path operation has parameters, they will be reflected in the formation of the URL, as well as in the Curl command.

Swagger also formulates the response body, and the headers included in the response, as well as the status code.

Next, expand the `user()` function mapped to the other path operation. This path operation function is defined to receive two parameters from the URL path. These parameters are listed in the browser window as shown in Figure 2-7. Enter their values and click the Execute button.

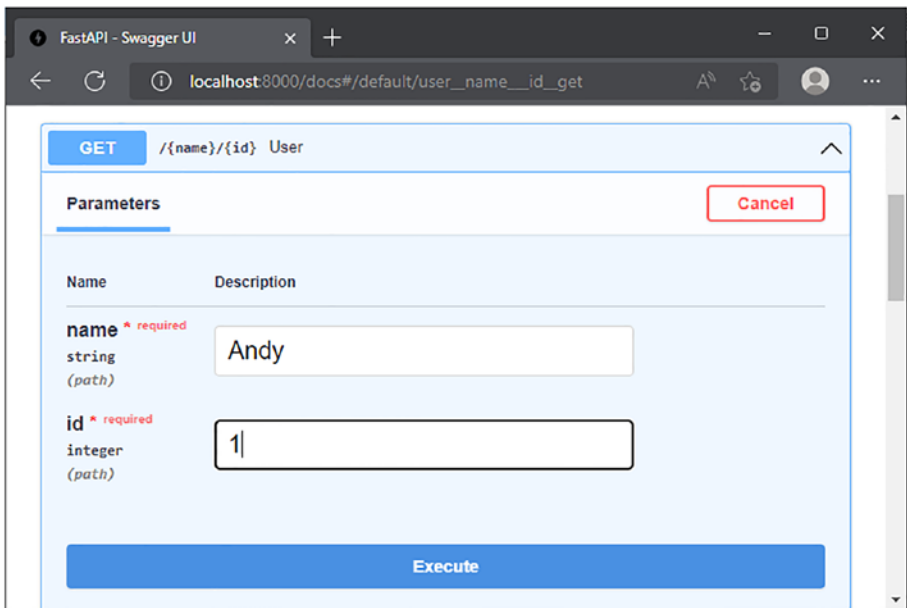


Figure 2-7. API docs of the function with parameters

The parameter values are appended to the fixed portion of the path, as shown in Figure 2-8.

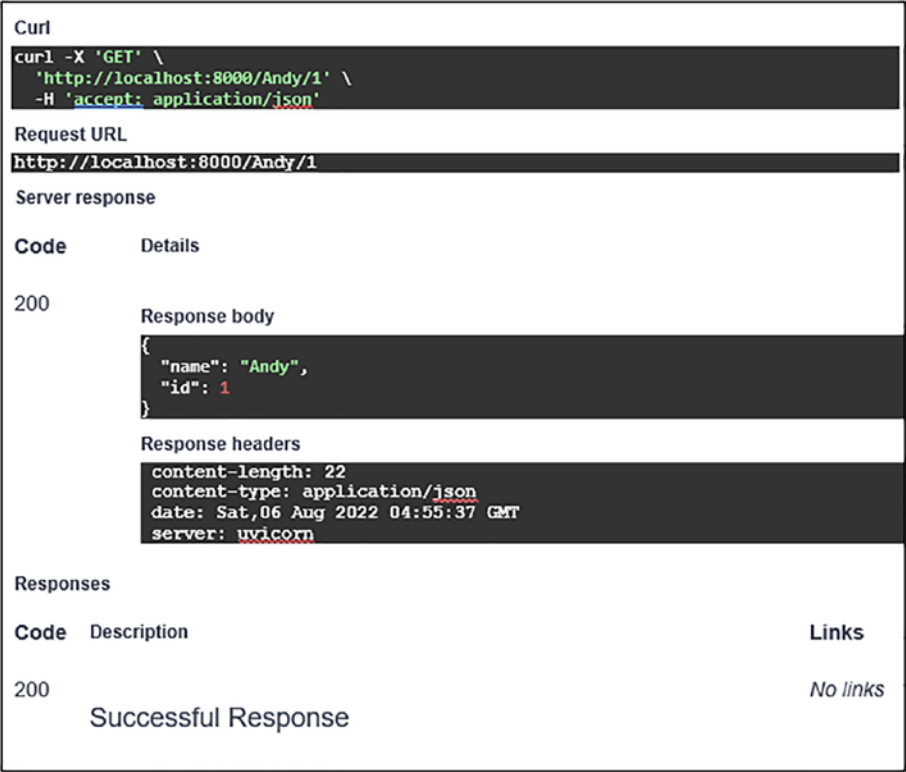


Figure 2-8. Server response showing parameter values

The JSON response of the server also includes the parameter values in the response body.

Redoc

Redoc is another open source tool from **Redocly** that generates the API documentation from the OpenAPI definitions. It has a responsive design, having a search bar, a navigation menu, the documentation, and examples of request and response.

Enter the URL `http://localhost:8000/redoc` (after starting the Uvicorn server) to display the Redoc documentation (refer to Figure 2-9). There are two path operations and functions in our code. These are shown along with the search bar at the beginning of the page.

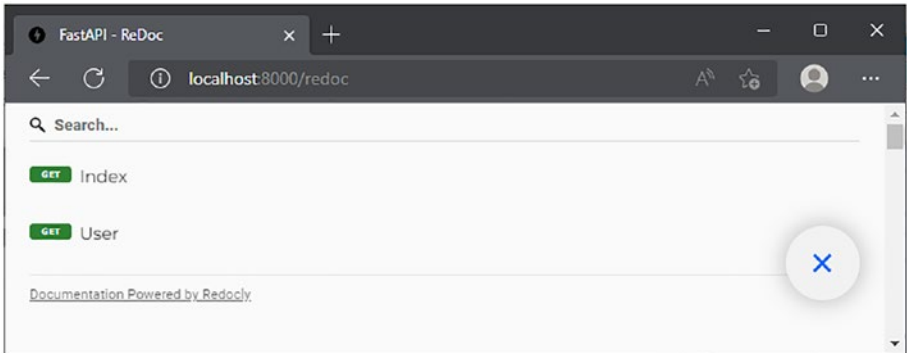


Figure 2-9. Redoc documentation showing a list of path operations

Click the User function to further explore the parameters of this function and its response (Figure 2-10).

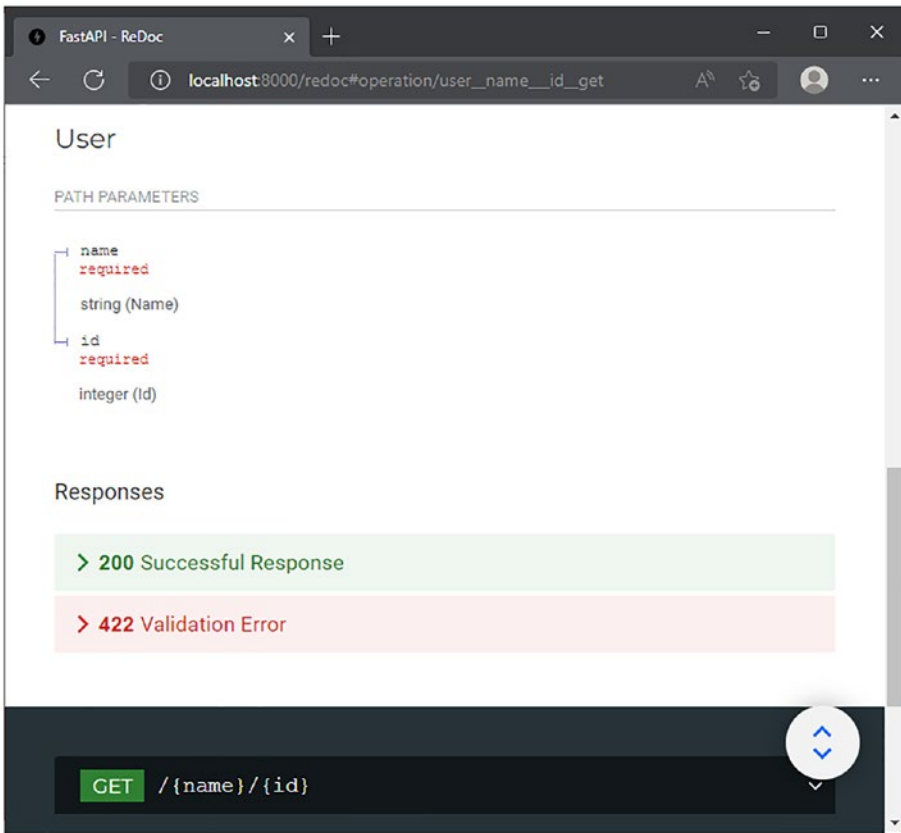


Figure 2-10. Parameters in Redoc docs

JSON Schema

The Swagger and Redoc tools actually translate the JSON representation of the API code. They use certain JavaScript and CSS code for an elegant representation of the raw JSON format in which the API schema is present. This representation is done in the “**application/schema+json**” media type. If you want to find out how the raw OpenAPI schema appears, use the URL <http://localhost:8000/openapi.json> in your browser. It displays the JSON data in Listing 2-7.

Listing 2-7. JSON Schema

```

{
  "openapi": "3.0.2",
  "info": {
    "title": "FastAPI",
    "version": "0.1.0"
  },
  "paths": {
    "/": {
      "get": {
        "summary": "Index",
        "operationId": "index__get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    }
  }
}

```

There are many API generation tools available, similar to Swagger and Redoc. You can easily configure FastAPI to use any of these tools. However, this is a slightly advanced maneuver and is beyond the scope of this book.

Path Parameters

In HTTP terminology, the path refers to the part of the URL that is trailing after the server's name or combination of (**IP address:port**). This path can be a mixture of fixed part and a variable part. Generally, the fixed part refers to a collection of resources available on the server, and the variable part (which may have one or more values) is used to locate or retrieve a specific resource from the collection.

Consider the request URL <http://mysite.com/employee/Rahul/20>. The intention is to retrieve the details of an employee with the name Rahul and having the age of 20. The **/employee** here is the fixed part, and the latter is the variable part consisting of two values. Obviously, they are likely to change for every request. How does FastAPI identify the variable parameters in the path and pass them to the operation function?

As already stated, the FastAPI application object acts as a router, directing the incoming request from the client to the appropriate handler function. It checks the request URL matches with the pattern declared in which operation decorator. Once the decorator is identified, its mapped operation function is executed, and the response is returned to the client.

If the path string of an operation decorator is expected to include one or more variable values, it has a placeholder for each. The placeholder is a valid Python identifier put inside curly brackets.

In Figure 2-11, for the request URL under consideration – <http://mysite.com/employee/Rahul/20> – the path string argument of the decorator has to be **/employee/{name}/{age}**. Whenever the client URL matches with this pattern, the data part is parsed into its respective variables and passed to the operation function below the decorator as path parameters, as shown in Figure 2-11.

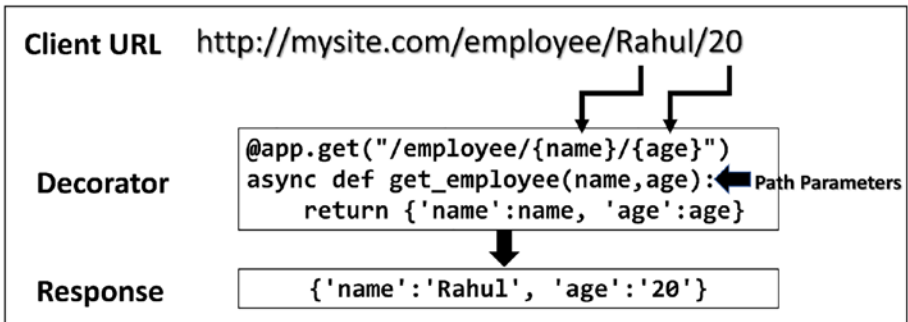


Figure 2-11. Parsing path parameters

Using Type Hints

Python parses the variable data from the URL into path parameters of string type by default. You can of course declare the type of a path parameter using the type annotation feature. The `get_employee()` coroutine in the preceding example is rewritten in Listing 2-8.

Listing 2-8. Parameters with type hints

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/employee/{name}/{age}")
async def user(name:str, age:int):
    return {"name":name, "age":age}
```

The Swagger documentation of this operation function (Figure 2-12) shows the two path parameters `name` and `age`, respectively, of `str` and `int` type.

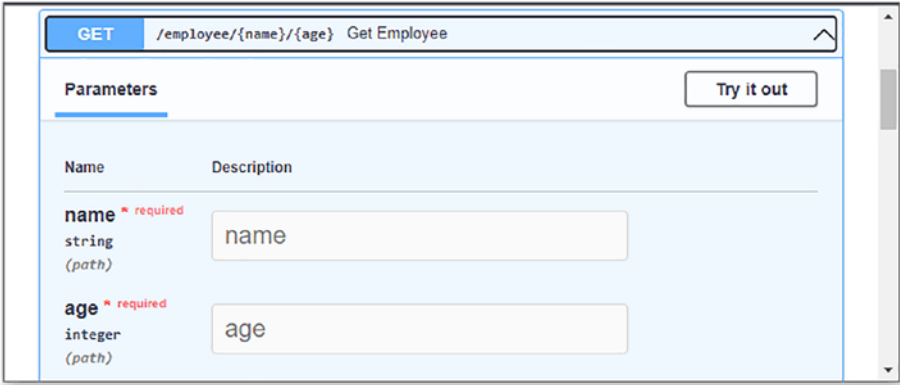


Figure 2-12. Swagger documentation showing type hints

Type Parsing

Since the parameters are defined with type hints, the values picked up from the URL are parsed to the suggested types whenever possible. For instance, in the preceding case, the trailing 20 in the URL is passed as an integer parameter to the `get_employee()` function. However, try entering the URL `http://localhost:8000/employee/Rahul/Kumar` in the browser's address bar, and you'll get the response shown in Listing 2-9.

Listing 2-9. JSON response with a conversion error message

```
{
  "detail": [
    {
      "loc": [
        "path",
        "age"
      ],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ]
}
```


Obviously, the error in this response is because the second parameter is not an integer, as declared in the operation function's definition.

Query Parameters

If a certain operation function (also called a view function) below the path decorator has some parameters in addition to the placeholders in the URL pattern, FastAPI deems them to be query parameters.

To understand this, let us rewrite the `@app.get()` decorator in the preceding example, as shown in Listing 2-10.

Listing 2-10. Function with path and query parameters

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/employee/{name}")
async def get_employee(name:str, age:int):
    return {"name":name, "age":age}
```

The decorator parses the value after **/employee** into a path parameter called `name`. However, its mapped function declares one more parameter – `age`. How does the router receive a value for this parameter?

Explore the Swagger documentation for the `get_employee()` function. Look at Figure 2-13.

The screenshot shows a REST client interface for a GET request to the endpoint `/employee/{name}` with the description "Get Employee". The "Parameters" tab is active. It lists two required parameters: "name" (string, path) with the value "Rahul" and "age" (integer, query) with the value "20". Both parameters are marked as "required" in red. A "Cancel" button is in the top right, and an "Execute" button is at the bottom.

Figure 2-13. Required parameters

In the list of parameters, while name is a path parameter as before, age is identified as a query parameter (note that both parameters are marked as **required**). Click the Execute button. Figure 2-14 shows the autogenerated request URL.

The screenshot shows the "Responses" tab of the REST client. It displays the generated curl command: `curl -X 'GET' \ 'http://localhost:8000/employee/Rahul?age=20' \ -H 'accept: application/json'`. Below this is the "Request URL" `http://localhost:8000/employee/Rahul?age=20`. The "Server response" section shows a status code of 200 and a JSON response body: `{ "name": "Rahul", "age": 20 }`. There are "Copy" and "Download" buttons for the response body.

Figure 2-14. URL with a query string

The URL `http://localhost:8000/employee/Rahul?age=20` shows the query string appended after a `?` symbol in front of the path parameter. If there are more than one query parameter, the parameter=value pairs are concatenated by the `&` symbol.

Optional Parameters

While path parameters are always required (the request URL must contain a value for each identifier in the URL pattern), query parameters may be declared to have a default value or may be set to be optional. In Listing 2-11, the query parameter `age` has a default value.

Listing 2-11. Parameter with a default value

```
@app.get("/employee/{name}")
async def get_employee(name:str, age:int=20):
    return {"name":name, "age":age}
```

The Swagger docs of this function (Figure 2-15) reflects the fact that the required mark on top of the `age` parameter is removed, and its default value is 20 (you can of course assign any other value if so required).

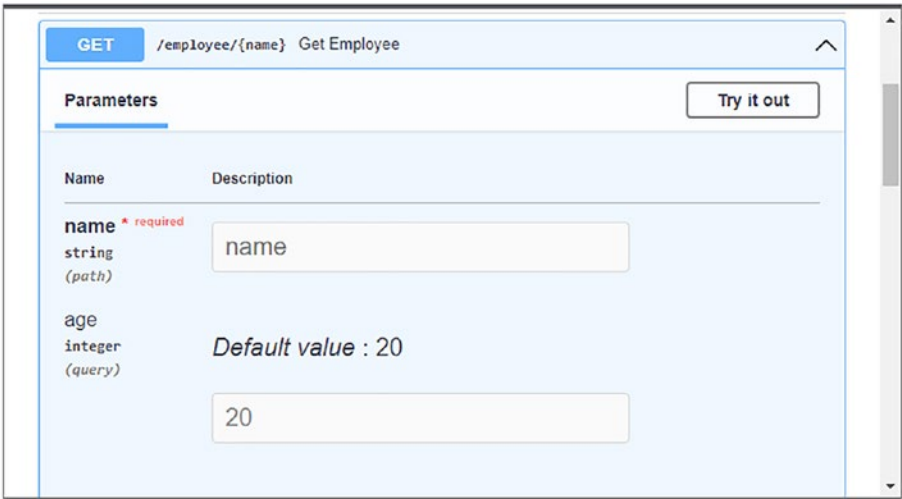


Figure 2-15. Query parameter with a default value

As a result, even if the query string is not given in the request URL (as in `http://localhost:8000/employee/Rahul`), FastAPI internally appends the query parameter, and the response sends its default value:

```
{
  "name": "Rahul",
  "age": 20
}
```

To let the client choose to give or not to give a value to any query parameter, use the `Optional` type from the `typing` module and set its default to `None`. In Listing 2-12, `age` is an optional parameter.

Listing 2-12. Optional parameter

```
from typing import Optional

@app.get("/employee/{name}")
async def get_employee(name:str, age:Optional[int]=None):
    return {"name":name, "age":age}
```

In such a case, the URL without a query string (`http://localhost:8000/employee/Rahul`) is perfectly acceptable, and the response shows a null value for the optional parameter:

```
{  
  "name": "Rahul",  
  "age": null  
}
```

Order of Parameters

So, we can now say that the path or the endpoint of a URL comprises a fixed part, one or more path parameters, and then followed by a query string that may have one or more query parameters. While the query string appears last in the path, the path parameters can be interspersed between two fixed parts. Take a look at Figure 2-16, showing URL parts.

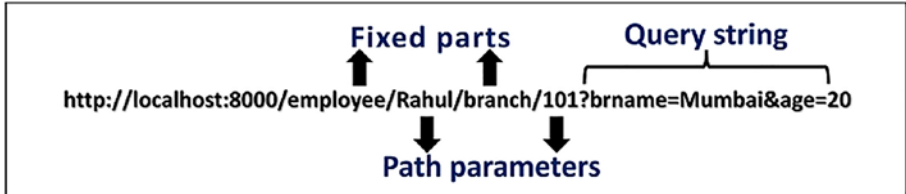


Figure 2-16. *Order of parameters*

Even though the path parameters precede the query parameters in the URL, it is not necessary to follow the order when declaring them in the definition of the operation function. In the code in Listing 2-13, the parameter declaration is quite random.

Listing 2-13. Order of parameters

```
from typing import Optional

@app.get("/employee/{name}/branch/{branch_id}")
async def get_employee(name:str, brname:str, branch_id:int,
                       age:Optional[int]=None):
    employee={'name':name, 'Branch':brname,
              'Branch ID':branch_id, 'age':age}
    return employee
```

The API documentation (Figure 2-17) too underlines the fact that the order of parameter declaration in the operation function need not be the same as in the URL path.

The screenshot displays a web interface for a FastAPI application. It features a form with four input fields, each with a label, a type, and a requirement status. The inputs are: 'name' (string, path, required) with value 'Rahul'; 'branch_id' (integer, path, required) with value '101'; 'brname' (string, query, required) with value 'Mumbai'; and 'age' (integer, query) with value '20'. Below the form are two buttons: 'Execute' (blue) and 'Clear' (white). Under the 'Responses' section, a cURL command is shown in a dark box: `curl -X 'GET' \ 'http://localhost:8000/employee/Rahul/branch/101?brname=Mumbai&age=20' \ -H 'accept: application/json'`. Below this, the 'Request URL' is shown in a dark box: `http://localhost:8000/employee/Rahul/branch/101?brname=Mumbai&age=20`.

Figure 2-17. *Order in declaration not important*

We see that irrespective of the declaration of parameters in the `get_employee()` function, the URL shows the query string trailing the path parameters.

Validation of Parameters

The path and query parameter components of the URL are user inputs. Hence, it is important that their values satisfy certain predefined criteria before they are forwarded to the operation function, so that the server doesn't throw unwanted exceptions.

In all the code examples used so far in this chapter, the path and query parameters have been declared to be of standard Python types – we have used `int` and `str`, but we can use `float` and `bool` as well. However, the FastAPI operation function can have a path or query parameter as an object of the `Path` or `Query` class, respectively. Both these classes are available in the `fastapi` module. The advantage of using a `Path` or `Query` instance as a parameter is that one or more validation constraints can be applied and some additional metadata can be included so that the documentation becomes more meaningful.

The `Path` and `Query` objects can be instantiated by passing certain parameters to the corresponding constructors. Their first parameter is the default value – generally set to `None`. The rest of the parameters are all optional keyword parameters. They include numeric constraints, the maximum and minimum length of string parameters, and the metadata of path parameters to specify the title, description, alias, etc.

Validating String Parameter

For the path or query parameters of string type, you can apply `min_length` and/or `max_length` constraints to ensure that their character length is in the desired range.

Let us modify the `get_employee()` function (from the previous example) and apply length validation criteria on `name` and `brname` parameters. Listing 2-14 gives the modified definition.

Listing 2-14. Validation of length

```
from fastapi import FastAPI, Path, Query
from typing import Optional
app = FastAPI()

@app.get("/employee/{name}/branch/{branch_id}")
```



```

async def get_employee(branch_id:int,name:str=Path(None,
min_length=10), brname:str=Query(None, min_length=5, max_
length=10), age:Optional[int]=None):
    employee={'name':name, 'Branch':brname, 'Branch ID':branch_
id, 'age':age}
    return employee

```

The order of the parameters has been changed because the Python function requires the nondefault arguments to be declared before those with default value. The validation checks applied mean that the name (path parameter) should not be smaller than ten characters, and the length of brname (query parameter) should be between five and ten characters. Figure 2-18 shows the Swagger documentation of this function. It highlights these criteria.

The image shows a Swagger UI interface for a GET endpoint. The URL bar displays `/employee/{name}/branch/{branch_id}` with the title "Get Employee". Below the URL bar, there is a "Parameters" section with a "Try it out" button. The parameters are listed in a table-like format with columns for "Name" and "Description".

Name	Description
branch_id * required integer (path)	branch_id
name * required string (path) minLength: 10	name
brname string (query) maxLength: 10 minLength: 5	brname
age integer (query)	age

Figure 2-18. Validation criteria in API documentation

Expect an erroneous response from the server if the validation fails, as in the URL `http://localhost:8000/employee/John/branch/101?brname=Secunderabad&age=21`. The name is less than ten characters long, and the number of characters in the name of the branch is not between five and ten as desired. Listing 2-15 shows the error response of failed validation.

Listing 2-15. Length validation failed

```
{
  "detail": [
    {
      "loc": [
        "path",
        "name"
      ],
      "msg": "ensure this value has at least 10 characters",
      "type": "value_error.any_str.min_length",
      "ctx": {
        "limit_value": 10
      }
    },
    {
      "loc": [
        "query",
        "brname"
      ],
      "msg": "ensure this value has at most 10 characters",
      "type": "value_error.any_str.max_length",
      "ctx": {
        "limit_value": 10
      }
    }
  ]
}
```

```

    }
]
}

```

Validation with RegEx

A regular expression (or **RegEx**) is a sequence of characters that specifies a search pattern. These patterns are used by string-searching algorithms for “find” or “find and replace” operations on strings or for input validation. The **re** module bundled in Python’s standard library implements RegEx functionality.

The `Path()` and `Query()` functions in FastAPI allow a RegEx parameter to be defined for validation so that the string value of the path/query parameter can be checked against the specified search pattern.

Let us include the RegEx parameter in the `Path()` function to constrain the value of `name` to either begin with **J** or end with **h** (as in **John**, **Javed**, or **Prakash**). The modified definition of the `get_employee()` operation function is shown in Listing 2-16.

Listing 2-16. Path validation with RegEx

```

@app.get("/employee/{name}/branch/{branch_id}")
async def get_employee(branch_id:int, brname:str, age:int,
                        name:str=Path(None, regex="^[J]|[h]$")):
    employee={'name':name, 'Branch':brname, 'Branch ID':branch_
id, 'age':age}
    return employee

```

After starting the application, open the browser and enter `http://localhost:8000/employee/Amar/branch/101?brname=London&age=21` as the URL. Since the name neither starts from **J** nor does it end with **h**, then you get the error response shown in Listing 2-17.

Listing 2-17. Response with failed RegEx validation

```

{
  "detail": [
    {
      "loc": [
        "path",
        "name"
      ],
      "msg": "string does not match regex \"^[j]|[h]$\"",
      "type": "value_error.str.regex",
      "ctx": {
        "pattern": "^[j]|[h]$"
      }
    }
  ]
}

```

Validating Numeric Parameters

Validation checks can also be applied to path/query parameters with numerical values.

The following types of validation criteria can be specified in the `Path()` or `Query()` constructor:

- **gt:** Greater than
- **ge:** Greater than or equal
- **lt:** Less than
- **le:** Less than or equal

Let us restrict the `branch_id` value to be between 1 and 100. Similarly, the age of the employee is required to be in the range of 21–60. The `get_employee()` function needs to be modified as shown in Listing 2-18.

Listing 2-18. Numeric range validation

```
from fastapi import FastAPI, Path, Query
app = FastAPI()
@app.get("/employee/{name}/branch/{branch_id}")
async def get_employee(name:str, brname:str,
                       branch_id:int=Path(1, gt=0, le=100),
                       age:int=Query(None, ge=20, lt=61)):
    employee={'name':name, 'Branch':brname, 'Branch ID':branch_
id, 'age':age}
    return employee
```

To test if the validations are applied correctly, use the URL `localhost:8000/employee/Rahul/branch/101?brname=Mumbai&age=15`. In Listing 2-19, the server’s response clearly shows that both the numeric parameters fail to meet the criteria.

Listing 2-19. Response with numeric validation failure

```
{
  "detail": [
    {
      "loc": [
        "path",
        "branch_id"
      ],
      "msg": "ensure this value is less than or equal to 100",
      "type": "value_error.number.not_le",
      "ctx": {
```

```

        "limit_value": 100
    }
},
{
    "loc": [
        "query",
        "age"
    ],
    "msg": "ensure this value is greater than or
equal to 20",
    "type": "value_error.number.not_ge",
    "ctx": {
        "limit_value": 20
    }
}
]
}

```

Adding Metadata

The metadata-related properties of `Path()` and `Query()` constructors allow certain descriptive features to the API documentation. These properties do not have any influence on the validation process. They just add some additional details about the parameter – such as title, description, etc.

You can add a suitable title and some explanatory text as the description for the parameter. The `alias` property can be used if the placeholder identifier in the endpoint mentioned in the decorator is different from the formal argument in the function. In the example in Listing 2-20, the name as a path parameter is customized with these metadata properties.

Listing 2-20. Parameter metadata

```

from fastapi import FastAPI, Path, Query
app = FastAPI()
@app.get("/employee/{EmpName}/branch/{branch_id}")
async def get_employee(branch_id:int, brname:str,
                        name:str=Path(None,
                                     title='Name of Employee',
                                     description='Length not
                                     more than 10 chars',
                                     alias='EmpName',max_
                                     length=10),
                        age:int=Query(None,include_in_
                                     schema=False)):
    employee={'name':name, 'Branch':brname, 'Branch ID':branch_
            id, 'age':age}
    return employee

```

The URL pattern in the `@app.get()` decorator has `EmpName` as the placeholder to parse the value from the request URL. To match with it, the `name` parameter is given an alias name.

These metadata properties are shown in the Swagger documentation as in Figure 2-19.

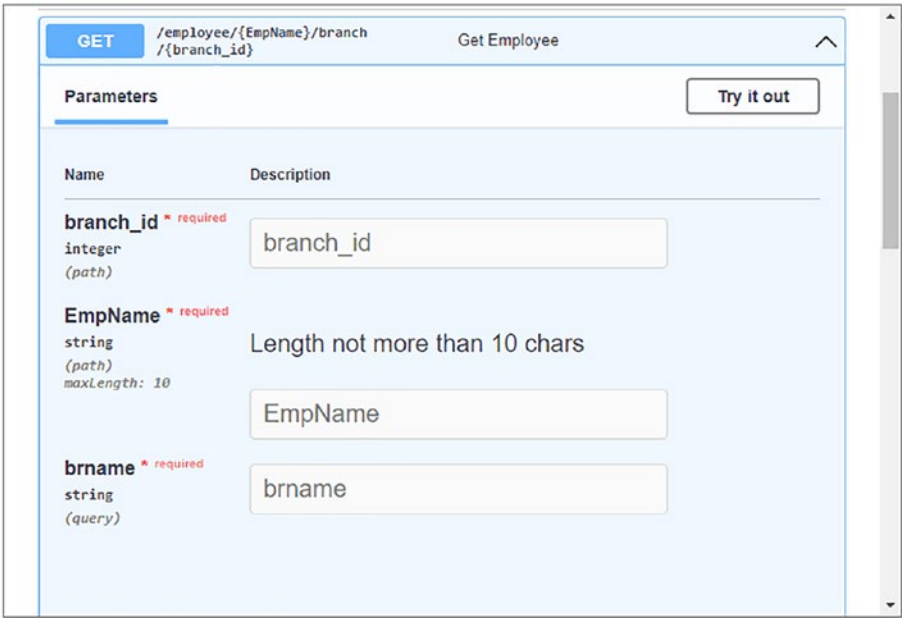


Figure 2-19. Metadata in Swagger documentation

Note that we have set the `include_in_schema` property for `age` – the query parameter – to `False`. As a result, it doesn’t appear in the docs of the function.

Summary

This chapter has helped you take the first steps toward learning to build a web app with FastAPI. We learned what its basic building blocks are. This chapter has also made you familiar with the API documentation tools – **Swagger** and **Redoc**.

In this chapter, the concept of path and query parameters has been explained in detail with the help of useful examples and the Swagger tool. In the end, we explored the provisions to perform parameter validation.

In the next chapter, we shall discuss another concept regarding inclusion of parameters within the body of the client request with the use of **Pydantic** models.

CHAPTER 3

Request Body

In the previous chapter, you learned how FastAPI handles the processing of path and query parameters included in the URL of the client's GET request. In this chapter, you will see how you can include required data as the body part of the client's HTTP request.

We shall cover the following topics in this chapter:

- POST method
- Body parameters
- Data model with Pydantic
- Model configuration
- Pydantic fields
- Validation
- Nested models

POST Method

A web browser software can send the request only through the GET method. We know that a GET request is used to retrieve one or more resources on the HTTP server. The path and/or query parameters in the request URL serve as a filter to fetch the data of specific resources.

Using the GET method for client-server interaction over HTTP has certain drawbacks. First of all, it is not very secure as the URL along with the parameter data is revealed in the address bar of the browser. Secondly, there is a limit to how much data can be sent to the server along with the GET request (and the limit is not very big either – in the range of a few kilobytes only!). Moreover, the data to be sent must be representable in ASCII characters only. That means any binary data such as an image can't be a part of the GET request.

To send a request for creating a new resource on the server, the HTTP protocol requires that the **POST** method should be used. The data that is required for a new resource is packed in the body of the POST request. This serves two purposes. The body part is not displayed in the browser's address bar; hence, it is a more secure method. Secondly, there is no size limit, and raw binary data can also be a part of the HTTP request body.

As a web browser cannot be used to raise a POST request directly, we have to find other means. We can use an HTTP client such as the **Curl** command-line tool to send a POST request. A typical example of Curl's POST command is shown in Listing 3-1.

Listing 3-1. Curl command for the POST method

```
curl -i -H "Content-Type: application/json" -X POST -d "{\"prodId\": \"1\", \"prodName\": \"Ceiling Fan\", \"price\": \"2000\", \"stock\": \"50\" }" http://localhost:8000/product
```

Note that we need to set the POST method explicitly with the **-X** option (remember that the default HTTP method is GET). The **-d** option is followed by the parameters and their values in JSON format. This data populates the body of the HTTP request.

We can also use certain web-based tools such as the **Postman** app or **Swagger UI** for this purpose or make an HTML form to send the request submitting the data with the POST method.

We have now become fairly conversant with the Swagger UI. We shall continue to use it in this chapter to understand how the data in the HTTP body is processed by FastAPI. In a subsequent chapter, we shall deal with the HTML form data.

Body Parameters

In a FastAPI app, POST requests are handled by the `@app.post()` decorator. As explained earlier, the path operation decorator needs a mandatory path string argument. (If the URL has any path parameters, their placeholder identifiers may appear in the path string as we learned in the previous chapter.)

The ASGI server passes the request's context data to the coroutine function (we call it an operation function) defined just below the `@app.post()` decorator. It contains the request object and the body data.

The value of each parameter of the body data is passed to the corresponding **Body** parameter declared in the operation function's definition. A Body parameter is an object of the Body class in FastAPI (similar to Path and Query classes).

To process the POST request raised by the Curl command mentioned earlier, let us define the `addnew()` operation function under the POST decorator (Listing 3-2).

Listing 3-2. POST operation function

```
from fastapi import FastAPI, Body, Request
app = FastAPI()

@app.post("/product")
async def addnew(request: Request, prodId:int = Body(),
prodName:str = Body(), price:float=Body(), stock:int = Body()):
    product={'Product ID':prodId, 'product name':prodName,
            'Price':price, 'Stock':stock}
    return product
```

The `addnew()` function is defined with the Body parameters – `prodId`, `prodName`, `price`, and `stock`. They are in fact the objects of the Body class. All the arguments to the Body class constructor are optional.

Run the preceding FastAPI code and launch the Uvicorn server on the localhost. Then open a command terminal and issue the POST Curl command mentioned earlier. As shown in Listing 3-3, the terminal returns a JSON response of the `addnew()` function along with the HTTP headers.

Listing 3-3. JSON response of the POST method

```
HTTP/1.1 200 OK
date: Fri, 26 Aug 2022 17:38:00 GMT
server: uvicorn
content-length: 71
content-type: application/json

{"Product ID":1,"product name":"Ceiling Fan","Price":2000.0,"Stock":50}
```

The Swagger UI is more convenient to use rather than the Curl tool, especially while testing the response of the routes of a FastAPI app (Figure 3-1). So, while the server is running, visit the `http://localhost:8000/docs` link with a web browser.

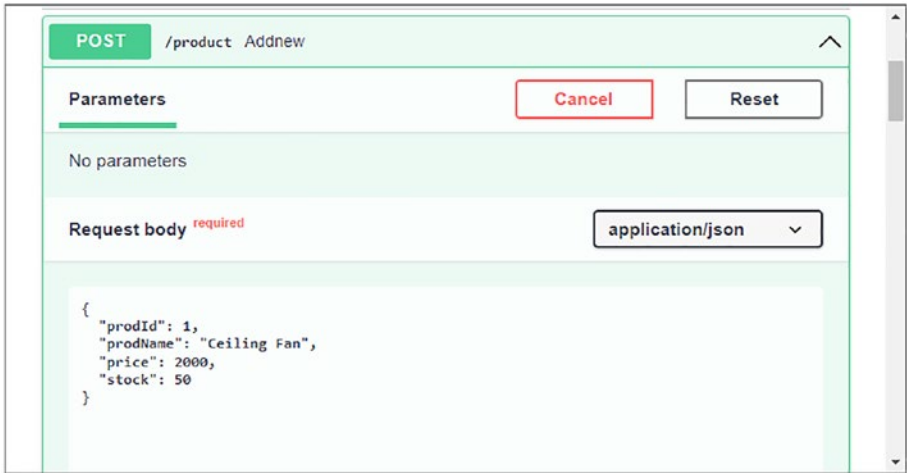


Figure 3-1. Body parameters in Swagger UI

Since there are no path or query parameters in the definition of the `addnew()` function, the parameter list is empty. But, the Request body section below it does show the body parameters. Assign them the values that we used in the Curl example and check the server's response. The Curl command construction, the headers, and the server's response are exactly the same as with the Curl command.

Parameters of the Body class constructor are similar to the ones we used for Path and Query constructors. In the previous chapter, we used the numeric and string validation criteria, as well as metadata parameters. They can also be used with the Body parameters.

Data Model with Pydantic

When a function definition has many parameters, it becomes very clumsy to pass that many arguments while calling. In the previous section, we defined the `addnew()` function with four Body parameters (`prodID`, `prodName`, `price`, and `stock`). In some other scenario, it could be more.

A workaround could be to declare a class (say **Product**) having properties as before and use its object as the parameter. However, it gives rise to another problem of validating the parameter values before processing them inside the function.

The **Pydantic** library addresses exactly the same problem. As we mentioned previously in the first chapter, Pydantic is a data modeling, validation, and parsing library. FastAPI makes extensive use of Pydantic, such as declaring the data model for populating the HTTP request body and efficiently performing **CRUD** operations on the databases (both SQL and NoSQL types).

dataclasses Module

At the center of the power of the Pydantic library is the `BaseModel` class. In a way, it is similar to the `dataclasses` library introduced in Python's standard library from version 3.7 onward.

The object of a Python class becomes a data container when decorated by the `@dataclass` decorator. It autogenerates the `__init__()` constructor for the user's class and also inserts the `__repr__()` method for the string representation of the object.

The `Product` class decorated by the `@dataclass` is declared as in Listing 3-4. Since there's an autogenerated constructor and string representation method in place, we can declare the object.

Listing 3-4. dataclass decorator

```
from dataclasses import dataclass
@dataclass
class Product:
    prodId:int
    prodName:str
    price:float
```

```

    stock:int
p1=Product(1, "Ceiling Fan", 2000, 50)
print (p1)

```

The `@dataclass` decorator also generates the magic methods like `__eq__()` for implementing Boolean operators and provides the dict as well as tuple representation of the class properties with `asdict()` and `astuple()` methods.

However, the `dataclasses` module doesn't have the mechanism of data validation. Hence, it is not possible to enforce schema constraints on the object data during runtime.

This is where Pydantic's `BaseModel` class comes into the picture.

BaseModel

At the center of the Pydantic library's functionality is the `BaseModel` class. A class that uses `BaseModel` as its parent works as a data container, just as a `dataclass`. Additionally, we can apply certain customized validation criteria on the properties of the class.

Listing 3-5 shows the basic usage of `BaseModel`. Let's declare the `Product` model, based on the `BaseModel`.

Listing 3-5. Product model

```

from pydantic import BaseModel

class Product(BaseModel):
    prodId:int
    prodName:str
    price:float
    stock:int

```

This class inherits the `schema_json()` method from `BaseModel` that renders its JSON representation, as shown in Listing 3-6.

Listing 3-6. Product schema

```
{
  "title": "Product",
  "type": "object",
  "properties": {
    "prodId": {
      "title": "Prodid",
      "type": "integer"
    },
    "prodName": {
      "title": "Prodname",
      "type": "string"
    },
    "price": {
      "title": "Price",
      "type": "number"
    },
    "stock": {
      "title": "Stock",
      "type": "integer"
    }
  },
  "required": [
    "prodId",
    "prodName",
    "price",
    "stock"
  ]
}
```


Before we explore the validation feature of Pydantic, let us see how it influences the FastAPI code.

Pydantic Model As Parameter

Let us declare a parameter of Product type in the `addnew()` operation function in the FastAPI code. Look at the script in Listing 3-7.

Listing 3-7. Using a Pydantic model as a parameter

```
from fastapi import FastAPI
from pydantic import BaseModel

class Product(BaseModel):
    prodId:int
    prodName:str
    price:float
    stock:int

app = FastAPI()
@app.post("/product/")
async def addnew(product:Product):
    return product
```

The moment FastAPI finds that the operation function has a Pydantic model parameter, the request body is populated by the properties in the model class – in our case, the Product class. The class specifications also help Swagger UI to generate the Product schema. Figure 3-2 shows the schema part of the documentation.

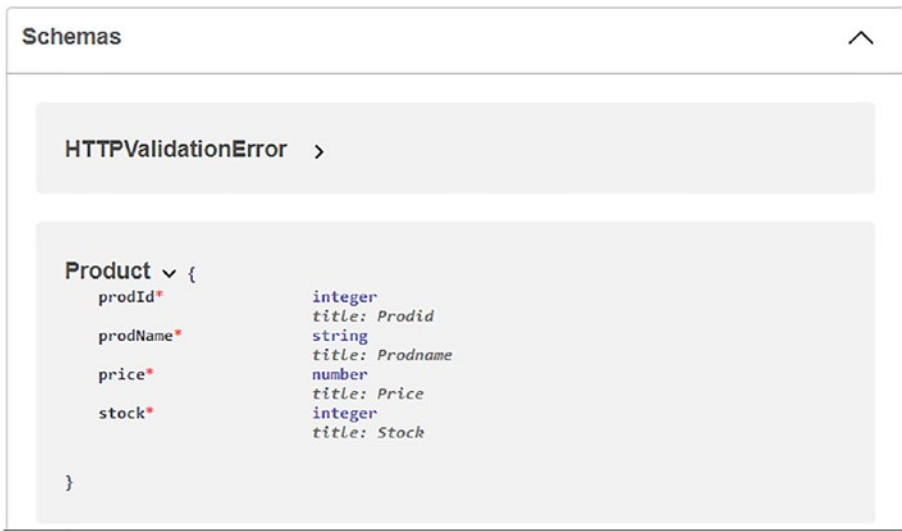


Figure 3-2. Product schema in Swagger

Typically, the POST method is used to add a new resource in the collection. Let us therefore maintain a list (`productlist`) of all the Product objects, as done in Listing 3-8. Every time the POST operation is done, the object is appended in the list. Accordingly, we need to modify the definition of the `addnew()` operation function.

Listing 3-8. POST operation function

```

productlist=[]

@app.post("/product/")
async def addnew(product:Product):
    productlist.append(product)
    return productlist

```

Add a couple of Product objects using the web interface of Swagger UI and check the server's response which looks as in Figure 3-3.

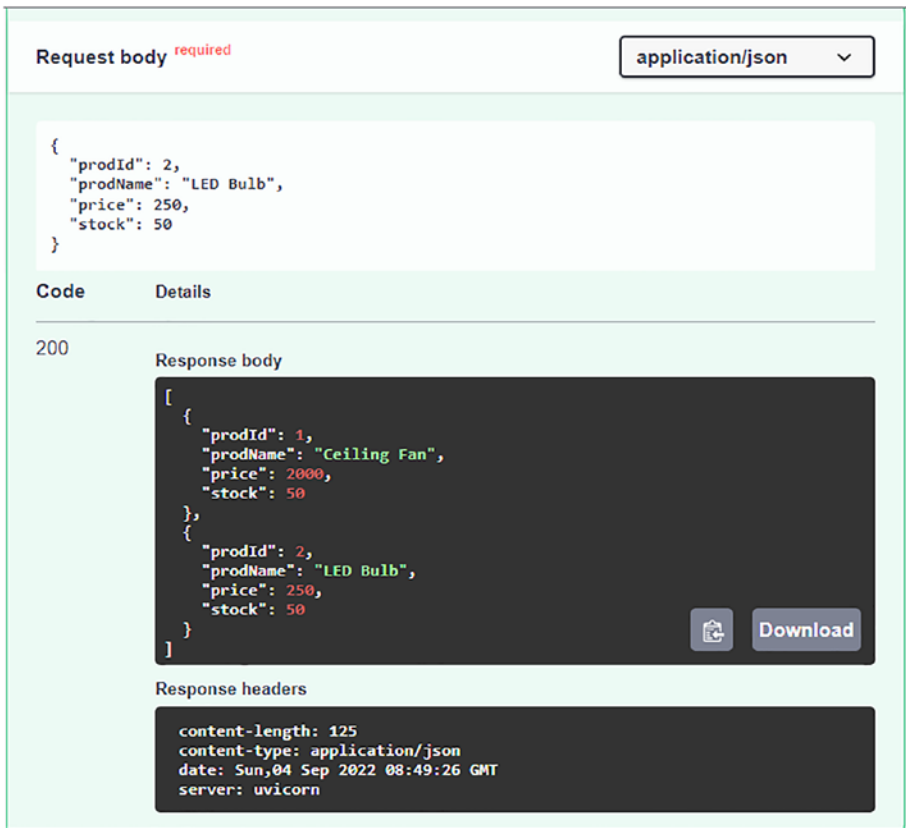


Figure 3-3. Server response with the Pydantic model

Once the model is passed, its attributes can be accessed and modified inside the operation function. Here, we would like to apply a tax of 10% on the price if it is greater than 5000. Listing 3-9 shows how the `addnew()` function modifies the price attribute.

Listing 3-9. Model attributes within a function

```
@app.post("/product/")
async def addnew(product:Product):
    dct=product.dict()
```

```

price=dct['price']
if price>5000:
    dct['price']=price+price*0.1
    product.price=dct['price']
productlist.append(product)
return productlist

```

Model Configuration

The `Config` attribute of the `BaseModel` helps in controlling the behavior of the model. It is in fact an object of the `BaseConfig` class. This configuration feature can be used in many ways. For example, the `max_anysr_length` decides what should be the maximum length for the model's string properties. You can also specify if you want the strings to always appear in upper- or lowercase. Set `anysr_upper` and/or `anysr_lower` to `True` if you want.

One of the cool `Config` settings is to include a `schema_extra` property (Listing 3-10). Its value is a dict object giving a valid example of the model object. This acts as additional information in the documentation of the JSON Schema of the model.

The `Product` model with `schema_extra` defined in its `Config` is rewritten in Listing 3-10.

Listing 3-10. Config class

```

from pydantic import BaseModel

class Product(BaseModel):
    prodId:int
    prodName:str
    price:float
    stock:int

```

```

class Config:
    schema_extra = {
        "example": {
            "prodId": 1,
            "prodName": "Ceiling Fan",
            "price": 2000,
            "stock": 50
        }
    }

```

When the Swagger interface generates the Product schema, this example data appears in it, as in Figure 3-4.

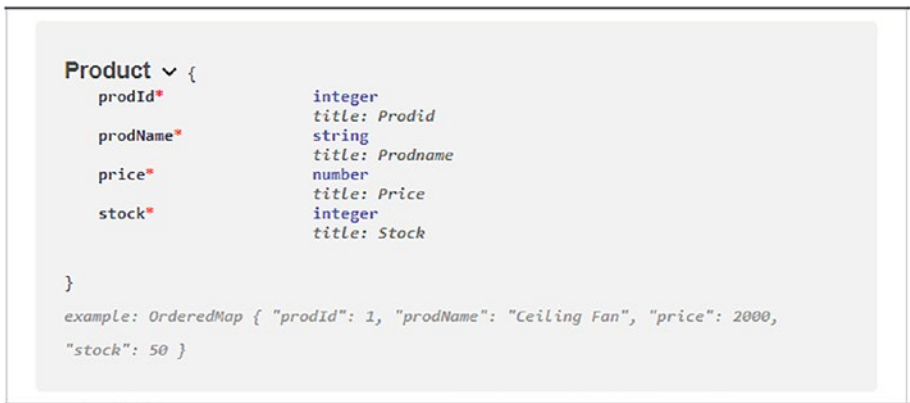


Figure 3-4. Example schema

orm_mode

The Config class inside the Pydantic model has an important property called **orm_mode**. If it is set to True, the Pydantic model can be created from any ORM model instance.

The term ORM stands for **object-relational mapper**. It is used for the programming technique of mapping a table structure in a SQL database

with a class declared in an object-oriented language such as Python.

SQLAlchemy is one of the most popular ORM libraries for Python. Its main advantage is that we can programmatically interact with the database and not by executing raw SQL queries.

If we set the `orm_mode` property to `True`, the model can be constructed from the instance of an ORM class such as the one inherited from SQLAlchemy's `declarative_base` class.

In our Product model, let us include the `orm_mode` configuration property (Listing 3-11).

Listing 3-11. ORM mode enabled

```
from pydantic import BaseModel
class Product(BaseModel):
    prodId:int
    prodName:str
    price:float
    stock:int
    class Config:
        orm_mode=True
```

Next, we shall declare a SQLAlchemy model (Listing 3-12) to match with the Pydantic model structure.

Listing 3-12. SQLAlchemy model

```
from sqlalchemy import Column, Integer, Float, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class ProductORM(Base):
    __tablename__ = 'products'
    prodId = Column(Integer, primary_key=True, nullable=False)
```

```

prodName = Column(String(63), unique=True)
price = Column(Float)
stock = Column(Integer)

```

The `from_orm()` method of Pydantic's `BaseModel` allows a model instance to be constructed from the ORM model object. Listing 3-13 shows the usage of this method.

Listing 3-13. Pydantic model from SQLAlchemy

```

prod_alchemy = ProductORM(
    prodId=1,
    prodName='Ceiling Fan',
    price=2000,
    stock=50
)

product = Product.from_orm(prod_alchemy)

```

Here, the `prod_alchemy` is initialized, and the same is used as an argument to the `from_orm()` method to obtain `product` as the object of the Pydantic model.

Conversely, the instance of the `Product` `BaseModel` can be parsed into an instance of the `ProductORM` model. We have a `dict()` method with the `BaseModel` that returns its dictionary representation. This dictionary is unpacked into an ORM object as in Listing 3-14.

Listing 3-14. SQLAlchemy model from Pydantic

```

product=Product(prodId=2, prodName='LED Bulb', price=250,
stock=50)
prod_alchemy=ProductORM(**product.dict())

```

We shall come back to this interfacing between the Pydantic and SQLAlchemy models later in this book when we discuss how to use a SQL database with the REST API.

Pydantic Fields

A model in FastAPI is simply a Python class inherited from Pydantic's `BaseModel`. Its class attributes become the fields in the model. The Product model used in the earlier section uses Python's built-in data types as type hints while declaring the fields in a model. The data types in the typing module consisting of type hints for collection data types can also be used. Hence, we can declare Pydantic fields to be of `typing.List`, `typing.Tuple`, and `typing.Dict`.

Let us define a Student model. The fields `StudentID` and `name` are of `int` and `str` types. In Listing 3-15, we have declared a `marks` field of `typing.Dict` type to store subject-wise marks.

Listing 3-15. Pydantic model with a Dict field

```
from pydantic import BaseModel
from typing import Dict

class Student(BaseModel):
    StudentID:int
    name:str
    subjects:Dict[str, int]
```

In the FastAPI code (Listing 3-16), the Student model is used as an argument to the POST operation function.

Listing 3-16. FastAPI app with the Student model

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Dict

class Student(BaseModel):
    StudentID:int
    name:str
```



```

    subjects:Dict[str, int]

app=FastAPI()

@app.post("/student")
async def addnew(student:Student):
    return student

```

The URL `http://localhost8000/student` can be tested in the Swagger tool's interface by entering values for the request body as shown in Figure 3-5.

POST /student Addnew

Parameters

No parameters

Request body **required** application/json

```

{
  "StudentID": 1,
  "name": "Daniel",
  "subjects": {
    "Maths": 90,
    "Physics": 89,
    "Comp Sci": 95
  }
}

```

Figure 3-5. Student model in the request body

The behavior of the Pydantic field can be customized by using the `Field()` function. It has the same properties as `Path()`, `Query()`, and `Body()` functions. For numeric fields, you can apply `gt`, `ge`, `lt`, `le` validation criteria. To restrict the length of a string field, there are `min_length` and `max_length` properties. The string pattern can be validated by

the `regex` property. You can follow the examples in the path parameter validation section to implement on Fields also.

In addition to Python's standard data types and the one in the typing module, the Pydantic library defines its own types.

HttpUrl: This field is essentially a string with built-in validation for URL schemes applied. The `HttpUrl` type accepts HTTP as well as HTTPS schemes. It requires a TLD (top-level domain) and host allowing a maximum length of 2083 characters.

The `AnyUrl` type allows any scheme (TLD is not required, but host required). The `FileUrl` field is for storing the file URL and doesn't require a host.

EmailStr: It is also a string and must be a valid email address. The validation of email representation of a string requires the `email-validator` module to be installed.

SecretStr: This data type is used mostly to store passwords or any other sensitive information that should not appear in logging or tracebacks. On conversion to JSON, it will be formatted as '*****'.

Json: Pydantic loads a raw JSON string, not the field of `Json` type. You can also use it to parse the loaded object into another type, based on the type `Json` is parameterized with.

Validation

The single most important feature of Pydantic is its capability to validate the input data before passing to the operation function. It has the built-in validation rules for all data types – standard types, those from the typing module, as well as Pydantic's own data types.

In the example (Listing 3-17), the `Employee` model implements the Pydantic data types explained earlier.

Listing 3-17. Pydantic types in a model

```

from fastapi import FastAPI
from pydantic import BaseModel, SecretStr, HttpUrl, Json
from typing import Dict

class Employee(BaseModel):
    ID: str
    pwd: SecretStr
    salary: int
    details: Json
    FBProfile: HttpUrl

app=FastAPI()

@app.post("/employee")
async def addnew(emp:Employee):
    return emp

```

The `addnew()` function is executed in the Swagger interface, with the request body populated with the data shown in Listing 3-18.

Listing 3-18. Sample Employee data

```

{
  "ID": "A001",
  "pwd": "asdf",
  "salary": 25000,
  "details":{"Designation": "Manager", "Branch":
"Mumbai"},
  "FBProfile": "https://www.facebook.com/dummy.employee/"
}

```

The `details` property is assigned a string containing a JSON with **Designation** and **Branch** as keys. Note that the internal quotation marks within the string have been escaped with the backslash (`\`) character.

The server's response, as in Listing 3-19, has the password field masked with asterisks.

Listing 3-19. Response after validation

```
{
  "ID": "A001",
  "pwd": "*****",
  "salary": 25000,
  "details": {
    "Designation": "Manager",
    "Branch": "Mumbai"
  },
  "FBProfile": "https://www.facebook.com/dummy.employee/"
}
```

If, however, the JSON value of the details field is erroneous, the server response indicates the reason (Listing 3-20), owing to Pydantic's built-in validation.

Listing 3-20. Error response of failed JSON validation

```
{
  "detail": [
    {
      "loc": [
        "body",
        "details"
      ],
      "msg": "JSON object must be str, bytes or bytearray",
      "type": "type_error.json"
    }
  ]
}
```

Similarly, if the URL validation fails, the server response is rendered with the corresponding error message. Have a look at Listing 3-21.

Listing 3-21. Response of failed URL validation

```
{
  "detail": [
    {
      "loc": [
        "body",
        "FBProfile"
      ],
      "msg": "invalid or missing URL scheme",
      "type": "value_error.url.scheme"
    }
  ]
}
```

Custom Validation

In addition to the built-in validations, you can provide your own. We need to use the `@validator` decorator in Pydantic.

In the Employee model used in the previous example, we have the ID field of string type. Let us impose a condition on its allowed value. It is desired that the string should have only the alphanumeric characters. If any character is non-alphanumeric, the validation should fail, and an appropriate error message should be generated (Listing 3-22).

Python's string class has the `isalnum()` method which returns false if any character in the string is non-alphanumeric. We use the same in the static function defined inside the Employee class. The method is decorated by the `@validator` decorator. Listing 3-22 shows the new version of the Employee model.

Listing 3-22. Using the @validator decorator

```

class Employee(BaseModel):
    ID: str
    pwd: SecretStr
    salary: int
    details: Json
    FBProfile: HttpUrl

    @validator('ID')
    def alphanum(cls, x):
        if x.isalnum() == False:
            raise (ValueError('Must be alphanumeric'))

```

To test the validation, give **#001** as the value of the ID field. The Unicorn server's response (Listing 3-23) throws the `ValueError` with the error message as mentioned in the preceding code.

Listing 3-23. Custom validation error message

```

{
  "detail": [
    {
      "loc": [
        "body",
        "ID"
      ],
      "msg": "Must be alphanumeric",
      "type": "value_error"
    }
  ]
}

```

Nested Models

As Pydantic models are in fact Python classes, we can always build a hierarchy of classes by defining a property (field) of a model as the object of an already defined model class. Listing 3-24 schematically represents the nesting of models.

Listing 3-24. Nesting of Pydantic models

```
ModelA:
```

```
..  
..
```

```
ModelB:
```

```
    ID:int  
    a1=ModelA  
    ..
```

The `a1` field of the `ModelB` class is the object of the `ModelA` class, which must be defined earlier.

Let us construct a more realistic nested structure of models. The `Products` model, in addition to the `ProductID`, `Name`, and `price`, has a `supplier` field. It is a list of objects of the `Suppliers` model (as more than one supplier may be dealing in a product).

So, we define the `Suppliers` model first as in Listing 3-25.

Listing 3-25. Suppliers model

```
from pydantic import BaseModel  
  
class Suppliers(BaseModel):  
    supplierID:int  
    supplierName:str
```

One of the fields of the Products model is a list of Suppliers (Listing 3-26).

Listing 3-26. Products model

```
from typing import List

class Products(BaseModel):
    productID:int
    productName:str
    price:int
    supplier:List[Suppliers]
```

Further, we create a model for Customers (Listing 3-27). In addition to the ID and name of the customer, the model defines a products field which in turn is a list of objects of the Products model. Let us use the top-level Customers model for populating the request body when the FastAPI receives a POST request.

Listing 3-27. Customers model

```
class Customers(BaseModel):
    custID:int
    custName:str
    products:List[Products]

from fastapi import FastAPI

app = FastAPI()

@app.post("/customer")
async def getcustomer(c1:Customers):
    return c1
```


The request body is filled with customer details. Let us say, two products are purchased. Details of each product are obtained from the Products model. Each product has a list of suppliers whose fields are fetched from the Customers model.

Use the example data in Listing 3-28 to test the **/customer** route that invokes the `getcustomer()` operation function.

Listing 3-28. Example data for the customers model

```
{
  "custID": 1,
  "custName": "C1",
  "products": [
    {
      "productID": 1,
      "productName": "P1",
      "price": 100,
      "suppler": [
        {
          "supplierID": 1,
          "supplierName": "S1"
        },
        {
          "supplierID": 2,
          "supplierName": "S2"
        }
      ]
    }
  ],
  {
    "productID": 2,
    "productName": "P2",
    "price": 200,
```

```
"supplier": [  
  {  
    "supplierID": 3,  
    "supplierName": "S3"  
  },  
  {  
    "supplierID": 4,  
    "supplierName": "S4"  
  }  
]  
}  
]
```

The OpenAPI schema based on all the models is autogenerated by Swagger UI (Figure 3-6). If we expand the Customers schema, it shows the nested relationship between the three models declared in this application.



Figure 3-6. *Customers schema*

Summary

This concludes our discussion of Pydantic models. We have learned how FastAPI uses Pydantic to populate the request body. We also explored the Pydantic field types and how to perform validations. An important feature of model configuration to enable its ORM mode has been explained here.

CHAPTER 3 REQUEST BODY

It will be very important when we'll extend our REST API application to connect with a database.

In the next chapter, we are going to study how certain dynamic data from the operation function is inserted into web templates, particularly the **Jinja** templates.

CHAPTER 4

Templates

FastAPI, as its name suggests, is primarily intended to be a library for API development. However, it can very well be used to build web apps of traditional type, that is, the ones rendering web pages, images, and other assets.

In this chapter, we shall learn how FastAPI renders web pages with the help of templates.

This chapter consists of the following topics:

- HTML response
- Template engine
- Hello World template
- Template with path parameter
- Template variables
- Serving static assets
- HTML form template
- Retrieve form data

HTML Response

By default, the response returned by the operation function is of JSON type. To be precise, an object of `JSONType` class is returned. However,

it is possible to override it by responses of different types such as `HTMLResponse`, `PlainTextResponse`, `FileResponse`, etc. All of them are classes derived from the `Response` class. Their type depends upon the `media_type` parameter in the constructor (Listing 4-1).

Listing 4-1. Response object

```
from fastapi import Response
resp=Response(content, media_type)
```

For example, by setting `media_type` to “**text/html**”, the operation function returns the content as the HTML response.

The `index()` function in the code in Listing 4-2 returns a string in HTML form. The string itself contains hard-coded HTML.

Listing 4-2. HTML response

```
from fastapi import FastAPI, Response

app = FastAPI()

@app.get("/")
async def index():
    ret=''
    <html>
    <body>
    <h2>Hello World!</h2>
    </body>
    </html>
    ...

    return Response(content=ret, media_type="text/html")
```

When the client browser visits the `http://localhost:8000` URL, it displays the Hello World string with HTML’s H2 style formatting.

As mentioned earlier, `HTMLResponse` is a subclass of the `Response` class. The content parameter in this case is a string (or byte string). This class has to be imported from the `fastapi.responses` module. Furthermore, we need to set it as the value of the `response_class` parameter of the `@app.get()` decorator.

The usage of the `HTMLResponse` class is demonstrated in the example in Listing 4-3. Here, we also define a path parameter – `name` – to the `index()` function. The idea is to parse it from the URL path so that the function renders the text **Hello Rahul** if `http://localhost:8000/Rahul` is the URL in the browser.

Listing 4-3. Parameter substitution in HTML response

```
from fastapi import FastAPI
from fastapi.responses import HTMLResponse

app = FastAPI()

@app.get("/{name}", response_class=HTMLResponse)
async def index(name):
    ret=''
    <html>
    <body>
    <h2 style="text-align: center;">Hello { }!</h2>
    </body>
    </html>
    ''.format(name)
    return HTMLResponse(content=ret)
```

The `format()` function of the `str` class inserts the value of the path variable at the placeholder inside the string. The HTML parser engine of the browser takes care of the tags and their attributes, thereby displaying the result as in Figure 4-1.



Figure 4-1. *HTML response in the browser*

Template Engine

HTML, the language used to construct web pages, is basically static in nature. Though we managed to add a certain interactivity in the preceding example by mixing the string representation of HTML code with a variable part, it is indeed a very cumbersome approach. Imagine how difficult it will be if you have to display a tabular data with some columns filled conditionally.

Most modern web application frameworks use a web template system for such a purpose. It inserts dynamically changing data items at appropriate places inside the well-designed static web pages. A web template system has three parts: a template engine, a data source, and a web template.

A web template is essentially a web page with one or more blocks of a certain template language code with the help of which the web page is populated with certain dynamic data items. On the other hand, the data to be interspersed in the web page is available in some data source in the form of a database table, a memory array, or a CSV file.

The template engine (also called template processor) receives these two parts. It fetches one set of data items at a time from the data source (such as one row from a database table) and puts them at their respective placeholders in the template to dynamically generate multiple web pages, one for each row in the data source. The schematic diagram in [Figure 4-2](#) explains the functioning of a template engine.

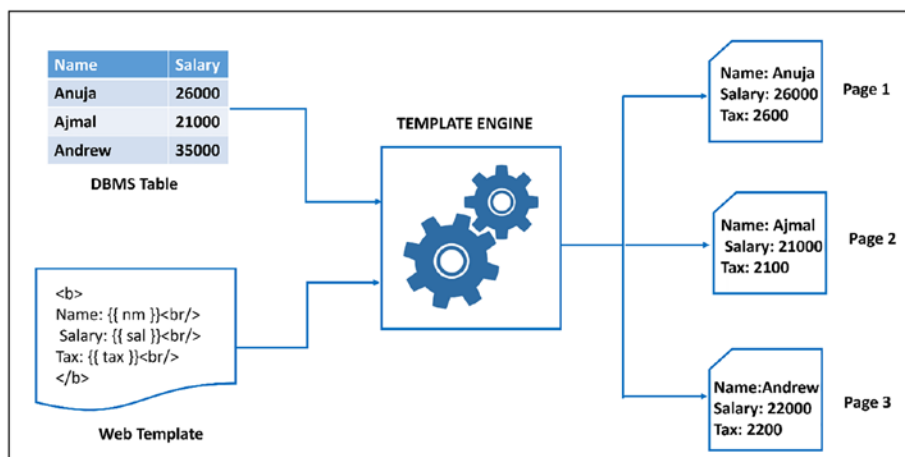


Figure 4-2. Template engine

Talking of templating in Python-based web apps, there are several templating languages available. Some frameworks have their own templating mechanism, while some others are bundled with a certain library. FastAPI doesn't have any such dependency on a particular template engine. In other words, a developer is free to use the library of their choice, although FastAPI recommends using the **jinja2** package. Since it is not installed by default, you need to do it explicitly, using the **pip3 install jinja2** command.

The jinja2 library is very popular. It is fast and lightweight. Because of its sandboxed environment, use of any potentially unsafe data is prohibited, and cross-site attacks are prevented. The template inheritance is a powerful feature in jinja2. It helps in maintaining uniformity in the web design.

Hello World Template

As mentioned earlier, a template is nothing but an HTML script. For the FastAPI app object to find the template, it must be placed in a templates folder, which should be in the same folder in which the FastAPI app object is present.

Earlier in this chapter, a hard-coded HTML string was rendered as the HTML response. Let us save the string contents as **hello.html** (Listing 4-4) in the templates folder alongside the app.py script.

Listing 4-4. Hello World template

```
<html>
  <body>
    <h2 style="text-align: center;">Hello World!</h2>
  </body>
</html>
```

FastAPI's support for jinja2 is available in the form of the Jinja2Templates function defined in the fastapi.templating module. So, we need to import it in our application code. This function returns a template object and needs the name of the directory in which the template web pages are stored (Listing 4-5).

Listing 4-5. Template object

```
from fastapi.templating import Jinja2Templates
template = Jinja2Templates(directory="templates")
```

As before, set the response_class attribute of the @app.get() decorator to HTMLResponse. The operation function needs to receive the request object parameter so that it can be passed to the template as a request context.

Call the `TemplateResponse()` method of the template object (refer to Listing 4-6). Its first parameter is the template web page to be rendered – `hello.html` in this case. The second parameter should be the request context. For now, it is just the request object received from the server. It returns the response to be sent to the client.

Listing 4-6. `TemplateResponse`

```
@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    return templates.TemplateResponse("hello.html", {"request":
        request})
```

Ensure that the application folder structure is as follows:

```
app/
|   main.py
|
└── templates/
    hello.html
```

Save the code (Listing 4-7) as `main.py`.

Listing 4-7. `main.py`

```
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse

from fastapi.templating import Jinja2Templates
template = Jinja2Templates(directory="templates")

app = FastAPI()

@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    return template.TemplateResponse("hello.html", {"request":
        request})
```

You should get the **Hello World** text rendered in the `<h2>` tag on the browser page as you run this application.

Template with Path Parameter

Although rendered as a template, the `hello.html` is really a static web page in the preceding example. The real power of the templating mechanism is to be able to render a dynamic content received from the renderer, that is, the view function.

The `TemplateResponse()` method's second argument is a dict object, often called a template context. It is mandatory that the request object is present in this context. In addition, the view function can include any number of objects in it. The keys in the context dictionary become the variables to be substituted in the HTML script.

The `jinja2` is a server-side templating library. The template web page uses various elements of the templating language as placeholders inside the HTML script. The template code is put in prescribed symbols to escape from the HTML content. (We shall come to know about the `jinja2` language syntax a little later.) The template engine reads the HTML script, inserts the context data in the `jinja2` placeholders, reassembles the HTML, and renders it to the client.

Earlier, we have seen how the path parameter is substituted in the string with HTML representation. Now, we shall include the path parameter (`name`) in the context dictionary. Modify the `index()` view function in the application code as in Listing 4-8.

Listing 4-8. Parameter in the request context

```
@app.get("/{name}", response_class=HTMLResponse)
async def index(request: Request, name:str):
    return template.TemplateResponse("hello.html", {"request":
        request, "name":name})
```

Use the context variable name and put it inside double curly brackets – **{{ name }}** – as the jinja2 placeholder in place of **World** in `hello.html` (Listing 4-9). The template engine substitutes it with the value of the `name` path parameter received at runtime.

Listing 4-9. Parameter substitution in the template

```
<html>
  <body>
    <h2 style="text-align: center;">Hello {{ name }}!</h2>
  </body>
</html>
```

As a result, after making these changes, running the Uvicorn server, and using `http://localhost:8000/Rahul` as the URL, the browser should show the **Hello Rahul** text as before.

Template Variables

As stated earlier, the view function inserts the request object and, if required, any other objects in the template context. The context itself is a `dict` object. Hence, the data to be passed to the template must have a string key.

The key in the context is treated as a template variable. In the preceding example, the “`name`” key (with its value being the path parameter `name`) is put in double curly brackets so that its runtime value is substituted in the HTML script.

You can add any Python object in the template context. It may be of standard data type (`str`, `int`, `list`, `dict`, etc.), an object of any other Python class, or even a Pydantic model. Let us have a look at some examples.

Passing dict in Template Context

You can assign a Python dictionary object as the value of a key in the template context. Inside the template code, you can use the `get()` method or the usual subscript (`[]`) syntax of the Python dict class to process the data received from the view.

Let us define an operation function in our application code (Listing 4-10) that receives two path parameters – name and salary. Instead of returning them in a JSON response, let us extend it to put them in a dictionary and pass it in the context of template. Add the code in `main.py` (it is currently running in debug mode) as shown in Listing 4-10.

Listing 4-10. dict as the template context

```
@app.get("/employee/{name}/{salary}", response_
class=HTMLResponse)
async def employee(request:Request, name:str, salary:int):
    data= {"name":name, "salary":salary}
    return template.TemplateResponse("employee.html",
    {"request": request, "data":data})
```

To display the values in the web page, obtain them with `data.get('name')` and `data.get('salary')` and put them within double curly brackets. Save the code (Listing 4-11) as `employee.html` in the templates folder.

Listing 4-11. Accessing dict in the template

```
<html>
  <body>
    <h1>Employee Details</h1>
    <h2>Name: {{ data.get('name') }} Salary: {{ data.
      get('salary') }}</h2>
  </body>
</html>
```

If the application is running with the debug mode on, change the URL to `http://localhost:8000/employee/Rahul/27`. The application's response in the browser should be as in Figure 4-3.

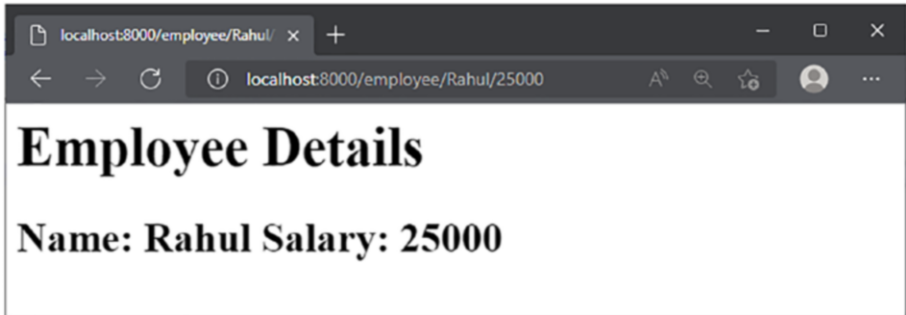


Figure 4-3. Web page with dictionary values

In addition to substituting template variables at runtime, you can include jinja2 statement blocks in the template for conditional processing as well as looping constructs.

Let us learn about the conditional statements of the jinja2 templating language.

Conditional Blocks in Template

In jinja2, there are three keywords to implement conditional logic inside the template. They are `if`, `elif`, and `else` – very much similar to those present in Python as well. We know that Python makes use of indentation to mark the blocks. A set of uniformly indented statements after the expressions involving `if`, `elif`, or `else` forms the conditional block. However, jinja2 doesn't use such indenting technique to mark the block. Instead, it uses the `endif` keyword to indicate the end of the block.

The conditional expressions are placed between `{% %}` symbols. They act as the delimiters so that the template processor is able to differentiate from the rest of HTML. The following schematic use of conditional statements in the template will help in understanding its implementation:

```
{% if expr1==True %}
    HTML block 1
{% elif expr2==True %}
    HTML block 2
{% else %}
    HTML block 3
{% endif %}
```

The logical expressions use the template variables as the operand. Of course, the HTML block may have one or more variable placeholders in double curly brackets where the context data is substituted.

In Listing 4-12, we have a jinja2 template web page in which `if - else - endif` statements are used.

Listing 4-12. Conditionals in the jinja2 template

```
<html>
  <body>
    <h1>Employee Details</h1>
    <h2>Name: {{ data.get('name') }} </h2>
    <h2>Salary: {{ data.get('salary') }}</h2>
    {% if data.get('salary')>=25000 %}
    <h2>Income Tax : {{ data.get('salary')*0.10 }}</h2>
    {% else %}
    <h2>Income Tax : Not applicable</h2>
    {% endif %}
  </body>
</html>
```


When rendered, the template shows the display (Figure 4-4) on the browser's page.

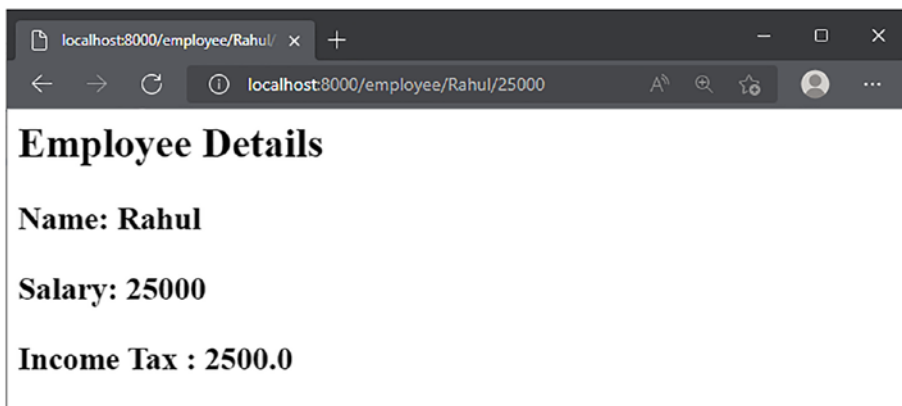


Figure 4-4. Conditional output in the template

Loop in Template

If any sequence object, that is, a list, tuple, or string, is passed as a context, it can be traversed inside the template with a looping construct. You have two keywords `for` and `endfor` for this purpose in `jinja2`. The syntax of the `for` statement is similar to the one in Python. Both the keywords, `for` and `endfor`, are put inside `{% %}` symbols. A typical `for` loop in `jinja2` code looks like this:

```
{% for item in sequence %}
HTML block
{% endfor %}
```

One can of course use a conditional block within the loop as well as construct nested loops.

In the operation function, a list object **langs** is passed in the template context. Save the code in Listing 4-13 as `main.py`.

Listing 4-13. List as the template context

```
@app.get("/profile/", response_class=HTMLResponse)
async def info(request:Request):
    data={"name":"Ronie", "langs":["Python", "Java", "PHP",
    "Swift", "Ruby"]}
    return template.TemplateResponse("profile.html",
                                    {"request":request,
                                    "data":data})
```

The value returned by `data.get('langs')` in the template is a list. A `jinja2` for statement is used to traverse it, and its items (names of programming languages) are rendered as HTML's unordered list elements.

Save the snippet in Listing 4-14 as **profile.html** in the templates folder.

Listing 4-14. Loop in the `jinja2` template

```
<html>
  <body>
    <h2>Name: {{ data.get('name') }} </h2>
    <h3>Programming Proficiency</h3>
    <ul>
      {% for lang in data.get('langs') %}
      <b><li> {{ lang }}</li></b>
      {% endfor %}
    </ul>
  </body>
</html>
```

The `http://localhost:8000/profile/` URL displays the output as shown in Figure 4-5.

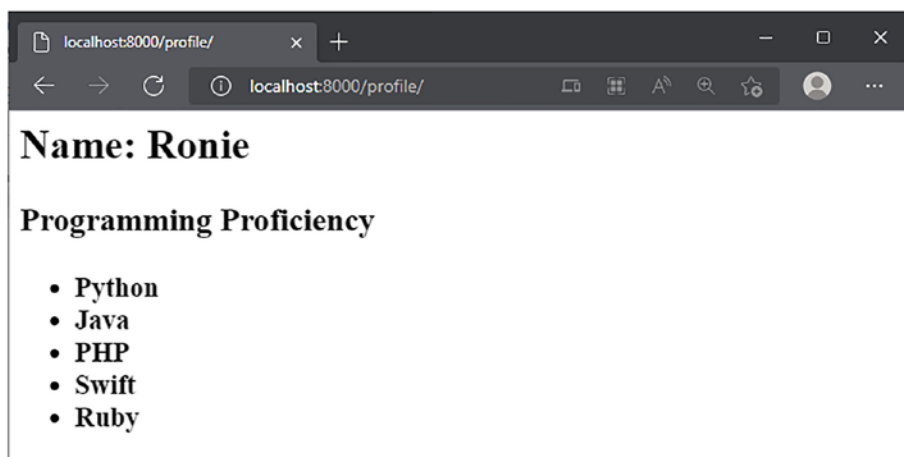


Figure 4-5. Browser output of the loop in the template

Serving Static Assets

The jinja2 template engine substitutes variable data in the placeholder tags to generate a web page dynamically. However, to display the content uniformly, HTML uses stylesheets. Similarly, HTML displays certain image files such as product images, logos, and icons. HTML also calls some JavaScript code for client-side processing. These resources – images, CSS stylesheets, and JavaScript files – don't change irrespective of the variable data. Hence, they are called static assets of a website.

FastAPI facilitates the use of such static files from a location configured for the purpose. These files should be put in a folder named static. It should be in the same folder where the application code script as well as the templates folder resides.

The path pointing to this static folder is mounted as a subapplication. The `mount()` method of the `Application` class defines a **/static** route mapped to its path:

```
app.mount("/static", StaticFiles(directory="static"),
name="static")
```

The `StaticFiles` class (imported from the `fastapi.staticfiles` module) maps the static directory to the `/static` route. By mounting a static subapplication, the static content becomes easy to deliver, process, and cache.

Whenever we want to use any of these static files, their path is obtained with the `url_for()` function provided by the `jinja2` template language.

Using JavaScript in Template

Normally, we find a JavaScript file included in the HTML script with the `src` tag:

```
<script src="path/to/myscript.js"></script>
```

However, referring to the file by its local path is a security concern. Instead, we use the `url_for()` function that will fetch the path during runtime. So we shall use the following syntax:

```
<script src="{{ url_for('static', path=myscript.js') }}"></script>
```

On the server side, include the operation function in the FastAPI application file (Listing 4-15).

Listing 4-15. View for the template with JS

```
from fastapi.staticfiles import StaticFiles
app.mount("/static", StaticFiles(directory="static"),
name="static")

@app.get("/testjs/{name}", response_class=HTMLResponse)
async def jsdemo(request:Request, name:str):
    data={"name":name}
```

```
return template.TemplateResponse("static-js.html",
                                {"request":request,
                                 "data":data})
```

As we know, the template web page (**static-js.html**) should be present in the templates folder. Listing 4-16 provides the required HTML script.

Listing 4-16. Calling JavaScript in the jinja2 template

```
<!DOCTYPE html>
<html>

<head>
    <title>My Website</title>
    <script src="{{ url_for('static', path=myscript.js') }}"></script>
</head>

<body>
    <h2>Using JavaScript in Template</h2>
    <h3> Welcome {{ data.get('name') }}</h3>
    <button onclick="myFunction()">Submit</button>
    <p id="response"></p>
</body>

</html>
```

This renders an HTML button, which when clicked calls a JavaScript function.

The **myscript.js** file in Listing 4-17 contains the definition of myFunction. Save the script in the static folder.

Listing 4-17. JavaScript function called by FastAPI

```
function myFunction() {  
    let text;  
    if (confirm("Do You Want to Continue\nChoose Ok/Cancel")  
        == true) {  
        text = "You pressed OK!";  
    } else {  
        text = "You pressed cancel";  
    }  
    document.getElementById("response").innerHTML = text;  
}
```

This function pops up the Confirm box. The user's response (Ok or Cancel) is placed in the paragraph element with **response** as its ElementId.

Start the Uvicorn server and enter `http://localhost:8000/testjs/Andy` in the browser, as shown in Figure 4-6. It displays the button.

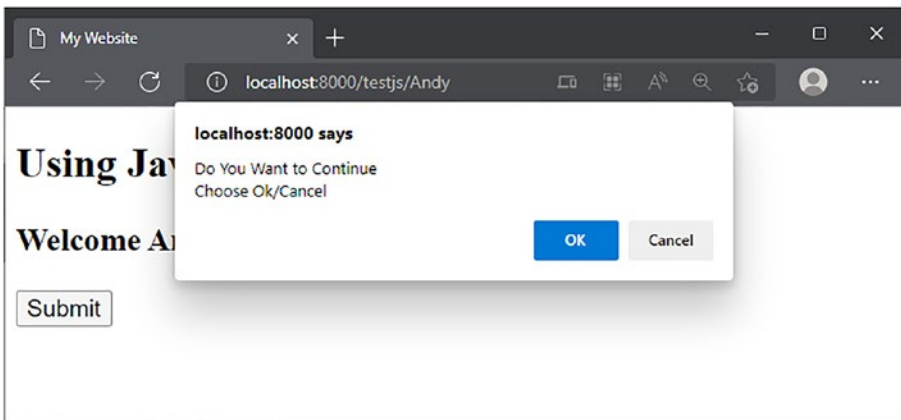


Figure 4-6. Confirm box

Assuming that the user clicks the OK button, the output is as shown in Figure 4-7.

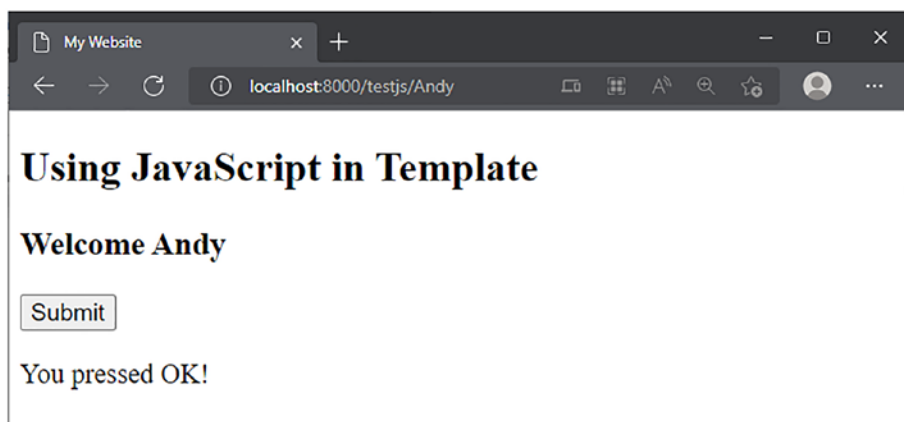


Figure 4-7. *Result of the Confirm box*

Static Image

To display an image in a web page, we normally use the `` tag with its usual syntax as follows:

```

```

However, as mentioned in the previous section, the local path to a file on the server should never be used as it is potentially a security risk. Instead, we use the `url_for()` function of Jinja2 to let the server retrieve the image path dynamically:

```

```

The FastAPI logo will be displayed on the web page. FastAPI expects the static images to be made available in the static folder. We need to add lines from Listing 4-18 as the operation function in our application code file.

Listing 4-18. View rendering a static image

```
@app.get("/img/", response_class=HTMLResponse)
async def showing(request:Request):
    return template.TemplateResponse("static-img.html",
                                     {"request":request})
```

Save the template script (Listing 4-19) as **static-img.html** in the templates folder.

Listing 4-19. jinja2 template for the static image

```
<!DOCTYPE html>
<html>

<head>
    <title>My Website</title>
</head>

<body>
    <h2 style="text-align: center;">Static image in
Template</h2>
    
</body>

</html>
```

The `http://localhost:8000/img/` URL displays the logo in the browser, as shown in Figure 4-8.

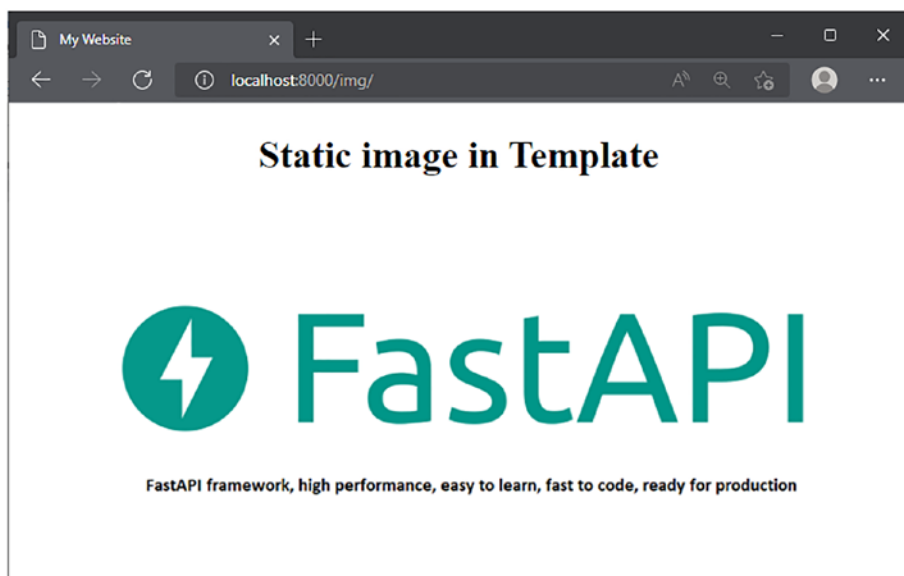


Figure 4-8. Static image

CSS As a Static Asset

Stylesheets are used for uniform presentation of various HTML elements across all the web pages in a website. For example, all paragraphs should use a font with the same size, color, etc. Or, the page body should have the same background color. Obviously, we are not going to discuss CSS in more detail here because it is beyond the scope of this book. We shall restrict ourselves to know how FastAPI serves the CSS files.

Again, in a normal HTML, a CSS file is linked with the following syntax:

```
<link href="mystyle.css" rel="stylesheet">
```

However, in a jinja2 template, we have to use the `url_for()` function to fetch the path of the CSS file:

```
<link href="{{ url_for('static', path='mystyle.css') }}"
rel="stylesheet">
```

We shall use the **profile.html** that we used to demonstrate the jinja2 loop. The unordered list is displayed with the link style defined in `mystyle.css`.

First, the updated `profile.html` is shown in Listing 4-20.

Listing 4-20. `Profile.html`

```
<html>
<head>
<link href="{{ url_for('static', path='mystyle.css') }}"
rel="stylesheet">
</head>
  <body>
    <h2>Name: {{ data.get('name') }} </h2>
    <h3>Programming Proficiency</h3>
    <ul class="b">
      {% for lang in data.get('langs') %}
      <b><li> {{ lang }}</li></b>
      {% endfor %}
    </ul>
  </body>
</html>
```

The script in Listing 4-21 is saved as a `mystyle.css` file and is placed in the **static** folder.

Listing 4-21. `mystyle.css`

```
h2 {
text-align: center;
}
```

```
ul.a {
    list-style-type: circle;
}

ul.b {
    list-style-type: square;
}
```

As shown in Figure 4-9, the bullet list of programming languages appears with a square symbol instead of the default circle.

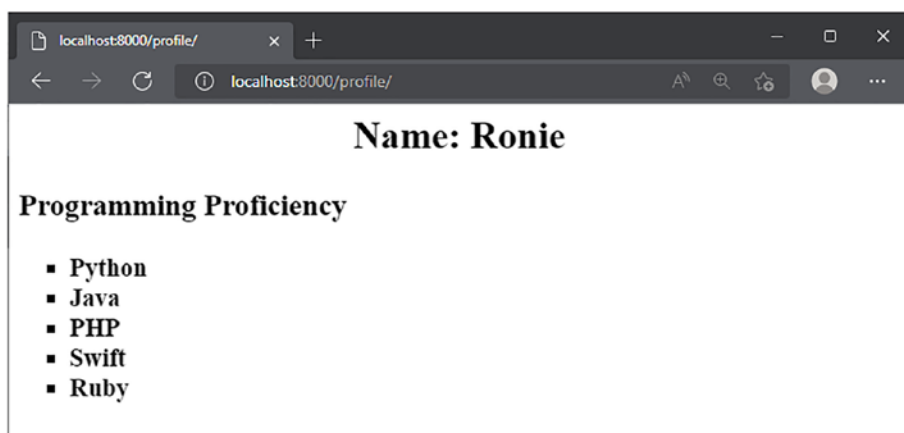


Figure 4-9. Using CSS in the jinja2 template

HTML Form Template

Often, a web application needs to present the user with a form to be filled and submitted for processing at the server end, when a certain URL is visited. Obviously, the template rendered by the corresponding operation function (or view) has to be an HTML form. Further, as the form is submitted, the data entered by the user should be retrieved by another view and processed accordingly.

To understand this workflow, let us use Listing 4-22 to design a simple form template (**form.html**) with two input elements and a dropdown.

Listing 4-22. form.html

```
<html>
<head>
<link href="{{ url_for('static', path='formstyle.css') }}"
rel="stylesheet">
</head>
<body>
<h3>Application Form</h3>

<div>
  <form action="/form/" method="POST">
    <label for="Name">Name of Applicant</label>
    <input type="text" id="name" name="name">

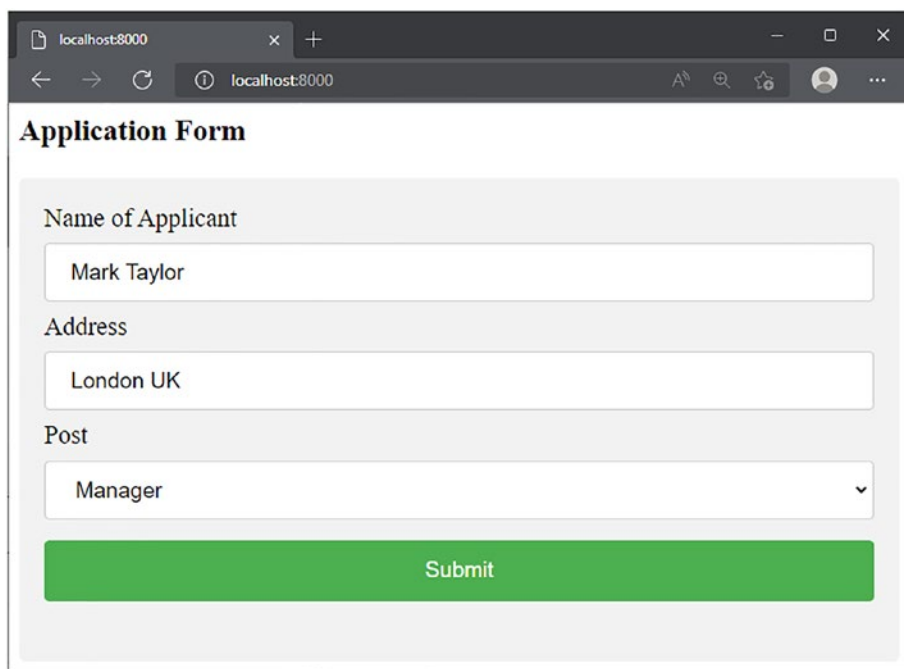
    <label for="Address">Address</label>
    <input type="text" id="add" name="add">

    <label for="Post">Post</label>
    <select id="Post" name="Post">
      <option value="Manager">Manager</option>
      <option value="Cashier">Cashier</option>
      <option value="Operator">Operator</option>
    </select>

    <input type="submit" value="Submit">
  </form>
</div>

</body>
</html>
```

This template is served when the browser requests the root URL `http://localhost:8000/` (Figure 4-10).



The screenshot shows a web browser window with the address bar set to `localhost:8000`. The page title is "Application Form". The form is styled with a light gray background and contains three input fields, each with a label above it. The first field, labeled "Name of Applicant", contains the text "Mark Taylor". The second field, labeled "Address", contains the text "London UK". The third field, labeled "Post", is a dropdown menu with "Manager" selected. Below the input fields is a large green button with the text "Submit".

Figure 4-10. HTML form

Retrieve Form Data

Note that this form is submitted to the `/form` URL path, with the POST request. Now we have to provide a view function to process the data submitted by the user. The view function receives the data in parameters that are objects of the `fastapi.Form` class. Moreover, the parameter names must match with the name attribute of each form element.

In the preceding HTML form, two text input elements have name attributes as “name” and “add,” while the Select element’s name attribute is “Post.” Hence, the view function mapped to the `/form` path (we shall

define the `getform()` function) should have these parameters. They must be objects of the `Form` class. Moreover, since the form's method attribute is `POST`, the operation decorator should be `@app.post()`.

Before we add the post decorator and its function, we need to install the **python-multipart** package. FastAPI needs it to process the forms:

```
pip3 install python-multipart
```

Import this package and add the `getform()` function in our application code (Listing 4-23). It parses the user's data in `Form` objects. The function returns a JSON response of the values in the form of a dictionary.

Listing 4-23. View with Form parameters

```
from fastapi import Form
@app.post("/form/")
async def getform(name:str=Form(...), add:str=Form(...),
Post:str=Form(...)):
    return {"Name":name, "Address":add, "Post Applied":Post}
```

You can test the `POST` operation with Swagger UI. Enter the test data as shown in Figure 4-11.

POST /form/ Getform

Parameters Cancel Reset

No parameters

Request body required application/x-www-form-urlencoded ▾

name * required
string

add * required
string

Post * required
string

Execute

Figure 4-11. Form parameters in Swagger UI

Click the **Execute** button and check the server's response (Figure 4-12).

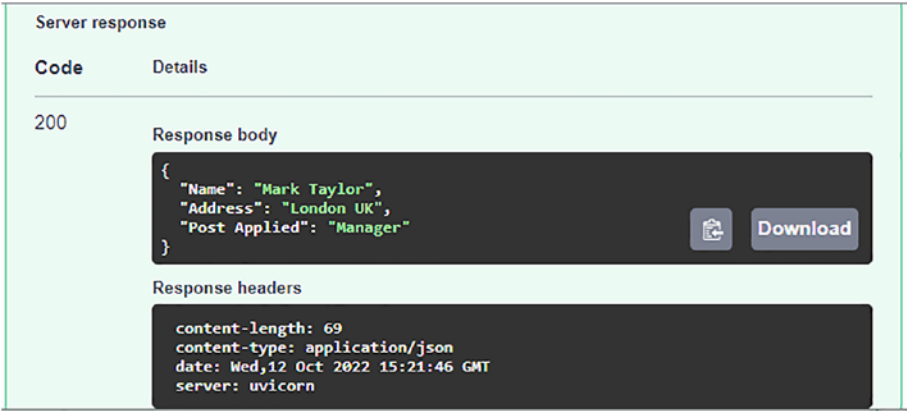


Figure 4-12. *Server response in Swagger*

If you want to check it on a live server, visit <http://localhost:8000/> to display the form. Enter the data and submit. Figure 4-13 shows the browser output.



Figure 4-13. *Response with Form data*

Summary

We hereby conclude this chapter on templates in FastAPI. We have learned how FastAPI renders the jinja2 template. Conditional and loop statements in the jinja2 template language have also been explained with appropriate examples. Lastly, we have discussed how to retrieve the data posted by an HTML form in a view function.

CHAPTER 5

Response

Any web application returns an HTTP response to the client. The operation function in a FastAPI app returns a JSON response by default. In this chapter, we shall learn how we can manipulate the response to handle different requirements.

In this chapter, the following points will be discussed:

- Response model
- Cookies
- Headers
- Response status code
- Response types

The FastAPI operation function *JSONifies* Python's primary types, that is, numbers, string, list, dict, etc., as its response. It can also return an object of a Pydantic model.

While the request object is passed to the operation function by the server itself, you need to formulate the response as a result of the function's process. The response object can also have `status_code`, `headers`, and `media_type` as additional parameters other than its content.

Response Model

You can declare any operation decorator with an additional `response_model` parameter so that the function adopts the response to the specified Pydantic model.

With the help of `response_model`, FastAPI converts the output data to a structure of a model class. It validates the data and adds a JSON Schema for the response in the OpenAPI path operation.

Consider the case of a Pydantic model called `Product` (Listing 5-1) having its field structure.

Listing 5-1. Product model

```
class Product(BaseModel):  
    prodId:int  
    prodName:str  
    price:float  
    stock:int  
    Inventory_val:float
```

The `Inventory_val` is a computed field in the preceding class. Hence, its value will not come from the POST request. The FastAPI can return the `Product` object, but let us say that we want to return only the Id, Name, and computed value of the inventory value. For this output structure, we declare another model as `ProductVal` (Listing 5-2).

Listing 5-2. ProductVal model

```
class ProductVal(BaseModel):  
    prodId:int  
    prodName:str  
    Inventory_val:float
```

We'll ask FastAPI to use this `ProductVal` model as its response type. To do this, use the `response_model` attribute as a parameter to the `@app.post()` decorator, and set it to our required model.

The client request brings the attributes of the `Product` class as the body parameters into the operation function. We'll have to compute the `Inventory_val` inside the function (Listing 5-3). As we return the `Product` object, the `response_model` is used to formulate the response; as a result, the price and stock attributes will not appear in it.

Listing 5-3. Using `response_model`

```
@app.post("/product/", response_model=ProductVal)
async def addnew(product:Product):
    product.Inventory_val=product.price*product.stock
    return product
```

Let us check the behavior with the help of Swagger UI. Enter the body parameters as shown in Figure 5-1.

POST /product/ Addnew

Parameters Cancel Reset

No parameters

Request body required application/json v

```
{
  "prodId": 1,
  "prodName": "Fan",
  "price": 2000,
  "stock": 25,
  "Inventory_val": 0
}
```

Figure 5-1. *Product model*

In Figure 5-2, we see that FastAPI formulates its response to the client as per the `response_model` set in the operation decorator.

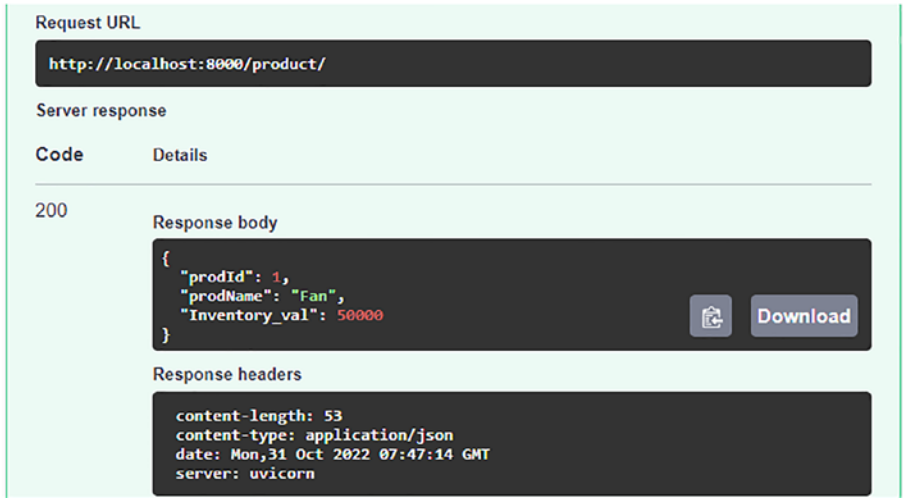


Figure 5-2. *Response model*

The `response_model` parameter is thus effective in limiting the output data to that of the model. You can further trim the response with the help of the following additional parameters for the operation decorator.

response_model_exclude_unset: If set to `True`, attributes in the input model with default values (to whom the client request doesn't have explicitly set values) will not appear in the response.

response_model_include: You can set this parameter to a list of only those attributes that you want to appear in the response.

response_model_exclude: The attributes listed will be omitted, including the rest in the response.

Cookies

When the server receives a request for the first time from any client browser, sometimes it may insert some additional data along with the response. This small piece of data is called a cookie. It is stored as a text file in the client's machine.

With the help of cookies, the web application keeps track of the user's activities, preferences, etc. On every subsequent visit by the same client, the cookie data already stored in its machine is also attached to the HTTP request body and the parameters. The cookie mechanism is a reliable method by which the web application can store and retrieve a stateful information regarding the client's usage, although HTTP is a stateless protocol.

Let us see how to set and retrieve cookies in a FastAPI application.

set_cookie() Method

The `set_cookie()` method of the Response object makes setting a cookie very simple. Let us understand this with an example. Have a look at Listing 5-4. Here's a simple login form (`form.html`) that sends the User ID and password to the server.

Listing 5-4. `form.html`

```
<form action="/setcookie" method="POST">
<h3>User ID</h3>
<p><input type="text" name="user"></p>
<h3>Password</h3>
<p><input type="password" name="pwd"></p>
<h3><input type="submit" name="submit"></p>
</form>
```

We want to store the User ID in the form of a cookie on the client's filesystem. Note that the form is being posted to the `/setcookie` URL path. We shall, therefore, provide an operation function for the `@app.post()` decorator.

This function reads the form data in two Form parameters – `user` and `pwd`. As the function returns its JSON response, we attach a cookie with the `set_cookie()` method of the Response object (Listing 5-5).

Listing 5-5. `set_cookie()` function

```
from fastapi.responses import Response
Response.set_cookie(key, value)
```

The collection of cookies is a dict object with key-value pairs. The `set_cookie()` method can also take `max_age` and `expires` as optional parameters.

The `setcookie()` function in Listing 5-6 sets a “user” cookie with the Form parameter `user` as its value.

Listing 5-6. Setting cookie

```
from fastapi import FastAPI, Request, Form
from fastapi.responses import Response

app = FastAPI()

@app.post("/setcookie/")
async def setcookie(request:Request, response: Response,
                    user:str=Form(...), pwd:str=Form(...)):
    response.set_cookie(key="user", value=user)
    return {"message":"Hello World"}
```

Cookie Parameter

How does the server read the cookies that come along with the client's request? The `fastapi` module defines the `Cookie` class. When its object is declared as one of the parameters of an operation function, the retrieved cookies are stored in it.

The `index()` function mapped to the `"/"` URL path has `user` as the cookie parameter. (The name of the parameter must be the same as the cookie previously set. If not, its value will be `None`.) This parameter is passed as the context to the HTML template (Listing 5-7).

Listing 5-7. Cookie parameter

```
from fastapi import Cookie
from fastapi.templating import Jinja2Templates

template = Jinja2Templates(directory="templates")

@app.get("/", response_class=HTMLResponse)
async def index(request: Request, user:str = Cookie(None)):
    return template.TemplateResponse("form.html",
                                     {"request":
                                      request,"user":user})
```

We need to add the template variable in the HTML script of our form. html so that it reads as in Listing 5-8.

Listing 5-8. Reading the cookie value

```
<html>
<body>
{% if user %}
<h3>You are logged in as {{ user }}</h3>
{% endif %}
<hr>
<form action="/setcookie" method="POST">
<h3>User ID</h3>
<p><input type="text" name="user"></p>
<h3>Password</h3>
<p><input type="password" name="pwd"></p>
```

```
<h3><input type="submit" name="submit"></p>
</form>
</body>
</html>
```

As a result, when the user visits “/” for the first time, the login form is displayed as in Figure 5-3.

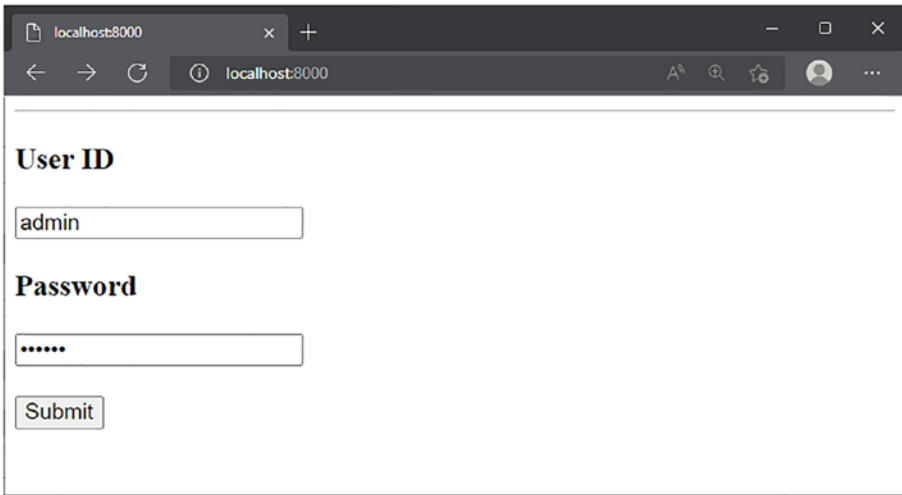
A screenshot of a web browser window. The address bar shows 'localhost:8000'. The page content includes a heading 'User ID' followed by a text input field containing the text 'admin'. Below this is a heading 'Password' followed by a password input field with masked characters '.....'. At the bottom of the form is a 'Submit' button.

Figure 5-3. *Login template*

Click the **Submit** button to post the form data to the `/setcookie` path. The operation function sets the user cookie. When the client revisits the “/” path, the form is rendered below the message showing the name of the user who is currently logged in. Figure 5-4 shows how it appears.

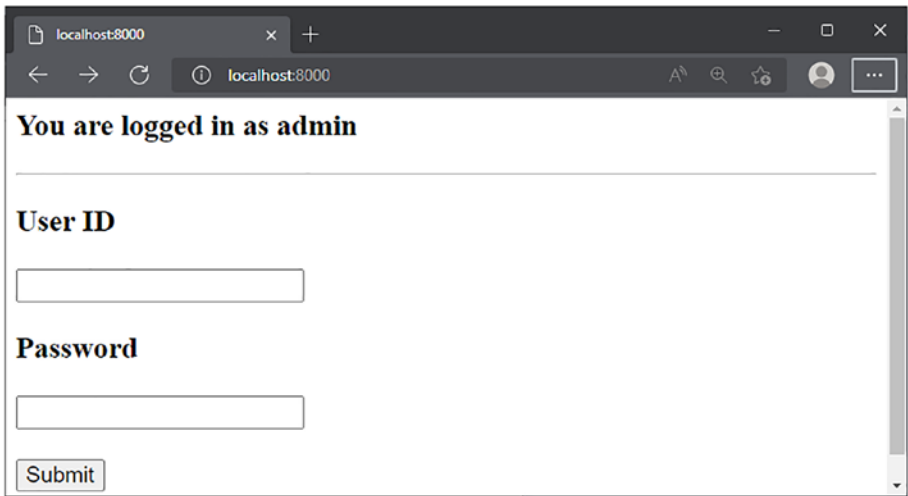


Figure 5-4. Login template showing the cookie value

Headers

Just as cookies, a web application may push a certain metadata in the form of HTTP headers into its response. In addition to the predefined HTTP header types, the response may include custom headers. To set a custom header, its name should be prefixed with “X”. In the example (Listing 5-9), the operation function adds a custom header called “**X-Web-Framework**” and a predefined header “**Content-Language**” along with the content to its response.

Listing 5-9. Setting a custom header

```
from fastapi.responses import JSONResponse
```

```
@app.get("/header/")
```

```
async def set_header():
```

```
    content = {"message": "Hello World"}
```

```
    headers = {"X-Web-Framework": "FastAPI", "Content-  
Language": "en-US"}
```

In response to the **/header** URL, the browser displays only the **Hello World** message. To check if the headers are properly set, you need to check the Swagger documentation of the `set_header()` function, as shown in Figure 5-5.

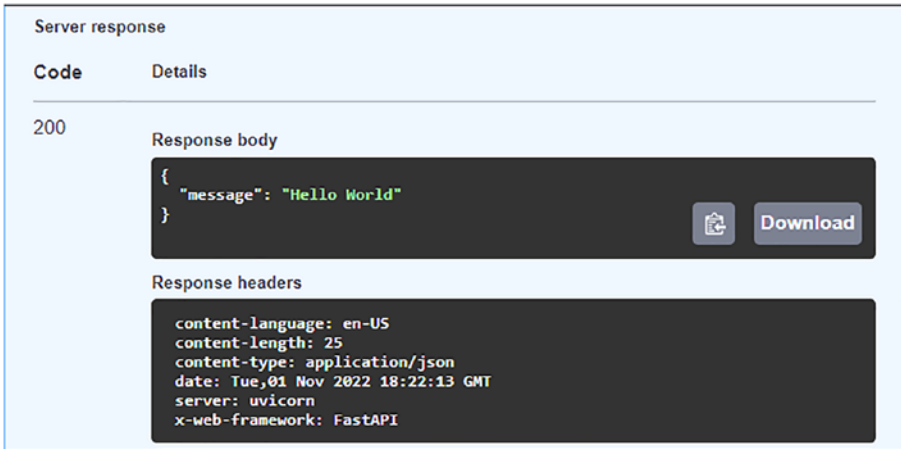


Figure 5-5. Response headers

The newly added headers will appear in the response headers section of the documentation.

Header Parameter

To read the values of an HTTP header from the client request, import the `Header` class from the FastAPI library, and use its object as a parameter in operation function definition. The name of the parameter should match with the HTTP header converted in `camel_case`. If you try to retrieve the “accept-language” header, “-” (dash) in the name of the identifier (since Python doesn’t allow it) is replaced by “_” (underscore).

As shown in Listing 5-10, add the decorator and its function in the application code.

Listing 5-10. Header parameter

```
from typing import Optional
from fastapi import Header
@app.get("/read_header/")
async def read_header(accept_language: Optional[str] =
Header(None)):
    return {"Language": accept_language}
```

You receive the value of the **accept-language** header in the response body (Figure 5-6).

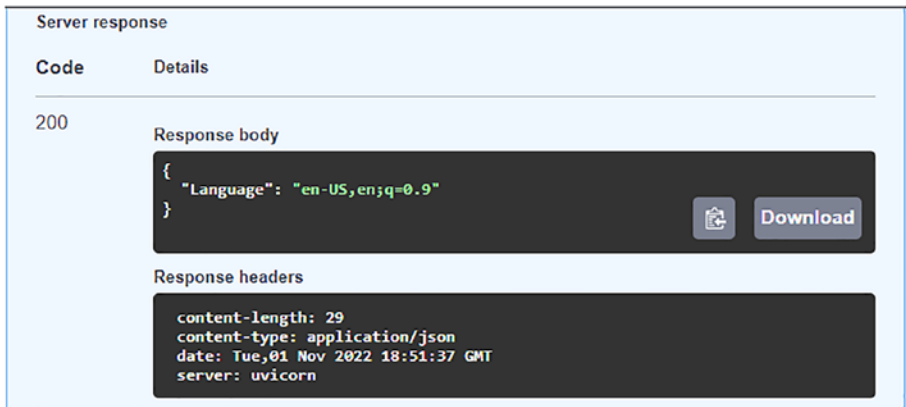


Figure 5-6. Header parameter

Response Status Code

In HTTP-based client-server communication, the server's response is accompanied by a numeric status code. It indicates how the client's request has been handled – whether it was successfully completed or whether the server encountered any problem.

The status code is a three-digit number. They have been categorized in five classes:

- **Informational responses:** Status codes between 100 and 199 are for “Information.” Normally, they are not set directly. Any response with 1XX status codes doesn’t have a body.
- **Successful responses:** Status codes starting with 2 indicate that the request has been successfully completed. 200 is the default status code. The code 201 is also common, usually after creating a new resource.
- **Redirection messages:** Status codes between 300 and 399 are used for redirection from one URL endpoint to another. For example, 301 is included in the response letting the user know that the URL of the requested resource has been changed permanently. 307 indicates temporary redirect.
- **Client error responses:** The 4XX codes imply client error responses. The most common one is 404, representing the page not found error.
- **Server error responses:** The codes starting with 5 represent the server errors. For example, 501 as the status code in the response means the server has encountered an internal error.

As mentioned earlier, the default status code is 200 OK. To include any other code in the response, use the `status_code` parameter in FastAPI’s operation decorator (Listing 5-11).

Listing 5-11. Adding a status code to the response

```
@app.get("/hello/{name}", status_code=201)
async def sayhello(name: str):
    return {"message": "Hello "+name}
```

The browser displays only the content part of the response. To verify the status code in the response, we need to check the Swagger documentation. Figure 5-7 shows the server response with a 201 status code.



Figure 5-7. Server response with a status code

FastAPI also defines status code constants corresponding to each numeric status code. For example, status code 200 is equivalent to `status.HTTP_200_OK`. In the code snippet (Listing 5-12), we use `status.HTTP_201_CREATED` as the value of the `status_code` parameter of the decorator.

Listing 5-12. Status code constants

```

from fastapi import status
@app.get("/hello/{name}", status_code=status.HTTP_201_CREATED)
async def sayhello(name: str):
    return {"message": "Hello "+name}

```

Response Types

As mentioned earlier, FastAPI returns its response in JSON form by default, by automatically converting the return value of the operation function with the help of `json-encoder`.

`JSONResponse` is a subclass of the `Response` class. You can directly return the object of the `Response` class, specifying the `media_type` such as `media_type="application/xml"`, `"text/html"`, etc.

For example, in [Listing 5-13](#), the function renders the response with the media type set as “text/html.”

Listing 5-13. Using `media_type`

```

from fastapi.responses import Response

app = FastAPI()

@app.get("/html/")
def get_html():
    data = """
<html>
<body>
<h3>Hello World</h3>
</body>
</html>
    """
    return Response(content=data, media_type="text/html")

```

To handle various types of responses, FastAPI provides different other subclasses of the `Response` class. We shall have a look at some of them.

HTMLResponse

The `HTMLResponse` class is derived from the `Response` class, with `media_type` set to “text/html.” So, in the preceding example, we can easily replace the return statement with this:

```
return HTMLResponse(content=data)
```

You may also typecast the response by setting the `response_class` parameter of the operation decorator. In Listing 5-14, `response_class` is set to `HTMLResponse`.

Listing 5-14. HTMLResponse

```
@app.get("/html/", response_class=HTMLResponse)
def get_html():
    data = """
    <html>
    <body>
    <h3>Hello World</h3>
    </body>
    </html>
    """

    return Response(content=data)
```

We have already used `HTMLResponse` as the `response_class` attribute very extensively for rendering the templates in the previous chapter.

Listing 5-15 is an example.

Listing 5-15. HTMLResponse as response_class

```
@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    return templates.TemplateResponse("hello.html", {"request":
        request})
```

JSONResponse

JSON is the default media type of FastAPI's response. If you are certain that data returned by the function is serializable with JSON, you can pass it directly to the response class and avoid the extra overhead (Listing 5-16).

Listing 5-16. JSONResponse

```
@app.get("/json", response_class=JSONResponse)
def get_html():
    data = "Hello World"
    return Response(content=data)
```

FastAPI also supports ORJSONResponse, a faster alternative JSON response, and UJSONResponse types.

StreamingResponse

This is a special type of Response object that takes either an async generator or a normal generator and streams its yield as the response.

In Python, a generator is a function that produces an iterator, and every time the yield statement is reached, the next value in the iterator is given out to the calling environment. Thus, it streams the series of values in the iterator.

In the example (Listing 5-17), the generator() function is a coroutine that yields a sequence of numbers, which are streamed as the application's response by the operation function.

Listing 5-17. Generator's StreamingResponse

```

async def generator():
    for i in range(1,11):
        yield "Line {}\n".format(i)

@app.get("/")
async def main():
    return StreamingResponse(generator())

```

On the client browser, you'll get the streaming line numbers displayed.

A disk file acts as a stream. Python's file object is an iterator. You can create a generator function that returns an iterator out of the file object. Every time the yield statement is executed, it streams the next line in the file. This is especially useful for bigger files as it is not necessary to read it all first in memory. Instead, pass the generator function to the StreamingResponse, and return it.

The code in Listing 5-18 has a readfile() generator that yields lines from the file. This generator is used by the StreamingResponse.

Listing 5-18. StreamingResponse from a file

```

file="large_file.txt"
@app.get("/")
def index():
    def readfile():
        with open(file, mode="rb") as f:
            yield from f

    return StreamingResponse(readfile(), media_
        type="text/plain")

```

You can even return the stream of MP4 video bytes as the response. Just change the media_type to **video/mp4**.

FileResponse

Note that the `open()` function that returns the file object doesn't support `async` and `await`. Hence, the operation function in the preceding example is not a coroutine but a normal function.

The `FileResponse` class is more suitable for streaming a file as the application's response. A few additional arguments may be given to instantiate the `FileResponse` object:

- **path:** The path to the file to stream.
- **headers:** You may include any custom headers if required.
- **media_type:** A string giving the media type. If not given, the media type is determined from the filename.
- **filename:** If set, this will be included in the Content-Disposition response.

In the example (Listing 5-19), the Uvicorn server streams the video content on the client browser.

Listing 5-19. FileResponse example

```
from fastapi.responses import FileResponse

file="wildlife.mp4"

@app.get("/", response_class=FileResponse)
async def index():
    return file
```

Start the server and visit the `http://localhost:8000/` URL. The video starts playing in the browser. A screengrab of the video is shown in Figure 5-8.

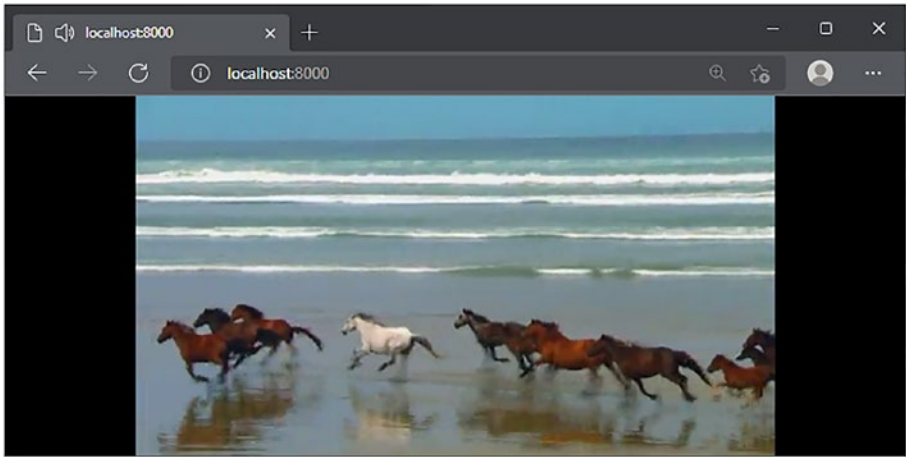


Figure 5-8. Streaming video with *FileResponse*

RedirectResponse

The mechanism of redirection in HTTP (called HTTP redirect) is a special kind of response. It is either used to redirect the user during site maintenance or downtime (this is a temporary redirect), or for Permanent redirect in situations like changing the site's URLs.

Redirect responses have status codes that start with 3. In FastAPI, the `RedirectResponse` class implements the HTTP redirect. By default, its status code is **307** – indicating a temporary redirect.

Let us see the use of `RedirectResponse` with the example in Listing 5-20.

Here, we have the `index()` function that renders a login form web page template. However, the form script is being modified, and hence the user needs to be prevented from accessing it and instead should be directed to another URL indicating that the page is unavailable.

The `index()` operation function is shown in Listing 5-20.

Listing 5-20. RedirectResponse example

```

from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.responses import Response, RedirectResponse
from fastapi.templating import Jinja2Templates

template = Jinja2Templates(directory="templates")

app = FastAPI()

@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    return RedirectResponse("/redirect")
    return template.TemplateResponse("form.html",
                                     {"request": request})

```

Note the call to `RedirectResponse` (in bold) with the URL (“**redirect**”) to which the original request is to be diverted. Listing 5-21 has the `redirected()` operation function.

The “**redirect**” is used inside the `@app.get()` decorator for the `redirected` operation function.

Listing 5-21. Redirect page

```

@app.get("/redirect")
def redirected(request: Request):
    return "The page you are looking for is temporarily unavailable"

```

As a result, when the client visits `http://localhost:8000/` with the intention of getting the login form, the application redirects it to the page showing the “temporarily unavailable” message.

You can create your own custom response class, as a subclass of the `Response` class. The `FastAPI` object can be instantiated with `default_response_class` to specify which type of response should be the default for the application.

Summary

In this chapter, you learned about the different types of responses that can be returned by the FastAPI application. The mechanism of including cookies and headers in the response was also explained in this chapter.

In the next chapter, you'll learn how to use databases in a FastAPI application.

CHAPTER 6

Using Databases

By now, you must have understood that the FastAPI application consists of various operation functions. Each operation function is invoked by the corresponding HTTP method decorator to which it is mapped. The HTTP methods POST, GET, PUT, and DELETE respectively create a resource, retrieve one or more resources available with the server, and update or delete one or more resources.

To perform persistent CRUD operations, the application needs to interact with a data storage and retrieval system. Applications generally use relational databases as a back end. Modern web applications, however, need databases capable of handling huge volume with dynamic schema, often called NoSQL databases. In this chapter, our objective is to understand how FastAPI interacts with relational and NoSQL databases.

The following topics are covered in this chapter:

- DB-API
- `aiosqlite` module
- SQLAlchemy
- `async` in SQLAlchemy
- PyMongo for MongoDB
- Motor for MongoDB

DB-API

A Python code can communicate with almost any type of relational database (such as SQLite, MySQL, PostgreSQL, Oracle, and many more). It needs a database-specific driver interface that is compatible with the DB-API standards. This ensures that the functionality of performing database operations is uniform for any type of database. As a result, only minimal changes will be required if the developer decides to change the back-end database.

To understand how FastAPI handles the database, we shall use **SQLite**. It is a lightweight and serverless database and has a built-in support in Python's standard library in the form of a `sqlite3` module, the reference implementation of DB-API. For the other relational databases though, you need to install the corresponding database driver, such as `mysqlclient` for MySQL or `Psycopg2` for PostgreSQL.

In this section, we shall build a FastAPI app to perform CRUD operations on the **Books** table in the SQLite database.

Creating the Books Table

The first step in the process is to establish a connection with the database and obtain the Connection object. Use the `connect()` function for this purpose (Listing 6-1).

Listing 6-1. Connect to SQLite

```
import sqlite3

conn=sqlite3.connect("mydata.sqlite3")
```

The string argument to the `connect()` function is the file representing the database. If **mydata.sqlite3** database doesn't already exist, it will be created.

To perform the database operations, we need a database cursor that handles all the query transactions (Listing 6-2).

Listing 6-2. Cursor object

```
cur=conn.cursor()
```

Call the `execute()` method on this cursor object. Its string argument holds the SQL query to be executed by the SQL engine. The Books table needed for our example is created by the code in Listing 6-3.

Listing 6-3. Creating the Books table

```
def init_db():
    conn=sqlite3.connect("mydata.sqlite3")
    cur=conn.cursor()
    qry=''
    SELECT count(name) FROM sqlite_master WHERE type='table'
    AND name='Books'
    ''
    cur.execute(qry)

    if cur.fetchone()[0]==0: #if the table doesn't exist
        qry=''
        CREATE TABLE IF NOT EXISTS Books (
            id INTEGER (10) PRIMARY KEY,
            title STRING (50),
            author STRING (20),
            price INTEGER (10),
            publisher STRING (20)
        );
        ''
```



```
        cur.execute(qry)
    conn.close()

init_db()
```

The `init_db()` function creates the Books table if it doesn't exist already. We have defined the title, author, and publisher as string fields and the price as an integer field.

How do you verify if the Books table is actually created? You can do it by more than one way. You can use any of the many SQLite GUI tools available (such as **SQLite Studio**). If you prefer to use **VS Code** for building the application, it is a good idea to install the **SQLite Viewer extension**. However, the easiest way is to open the SQLite shell (Listing 6-4), open the database, and check the schema of the Books table.

Listing 6-4. SQLite shell

```
sqlite> .open mydata.sqlite3
sqlite> .tables
Books
sqlite> .schema Books
CREATE TABLE Books (
    id INTEGER (10) PRIMARY KEY,
    title STRING (50),
    author STRING (20),
    price INTEGER (10),
    publisher STRING (20)
);
```

With the database now created, let us perform further operations on it.

Inserting a New Book

As mentioned earlier, the POST method is used to create a new resource, in this case adding a new book data in the Books table. To include the data in the request body, the Pydantic model is declared as in Listing 6-5.

Listing 6-5. The Book class as the Pydantic model

```
from pydantic import BaseModel
class Book(BaseModel):
    id: int
    title: str
    author: str
    price: int
    publisher: str
```

The POST operation function uses an object of the Book model as the parameter. In addition, this function will also need the database contexts – connection and cursor objects – so that the INSERT operation can be carried out. They can be seen as the dependencies of the operation function.

Note that the operation function is not explicitly called, but gets invoked when the matching URL is entered by the client. Hence, there should be some mechanism to pass the parameters (other than path, query, or body parameters) to be included into the operation function. The Depends() function helps inject the desired dependencies. (We shall take a detailed look at this concept later in the book.)

Let us define the get_cursor() function (Listing 6-6) that yields a tuple of connection and cursor object of our database.

Listing 6-6. Dependency function

```
from fastapi import Depends

def get_cursor():
    conn=sqlite3.connect("mydata.db")
    conn.row_factory = sqlite3.Row
    cur=conn.cursor()
    yield (conn,cur)
```

Now we are in a position to define our POST operation function. It uses the Pydantic model attributes and executes the **INSERT** query with the help of the cursor object available to it through the injected dependencies.

Listing 6-7 has an `add_book()` function that receives the request body as the Pydantic model object and uses its attributes in the parameterized INSERT query, so that a new row is added in the Books table.

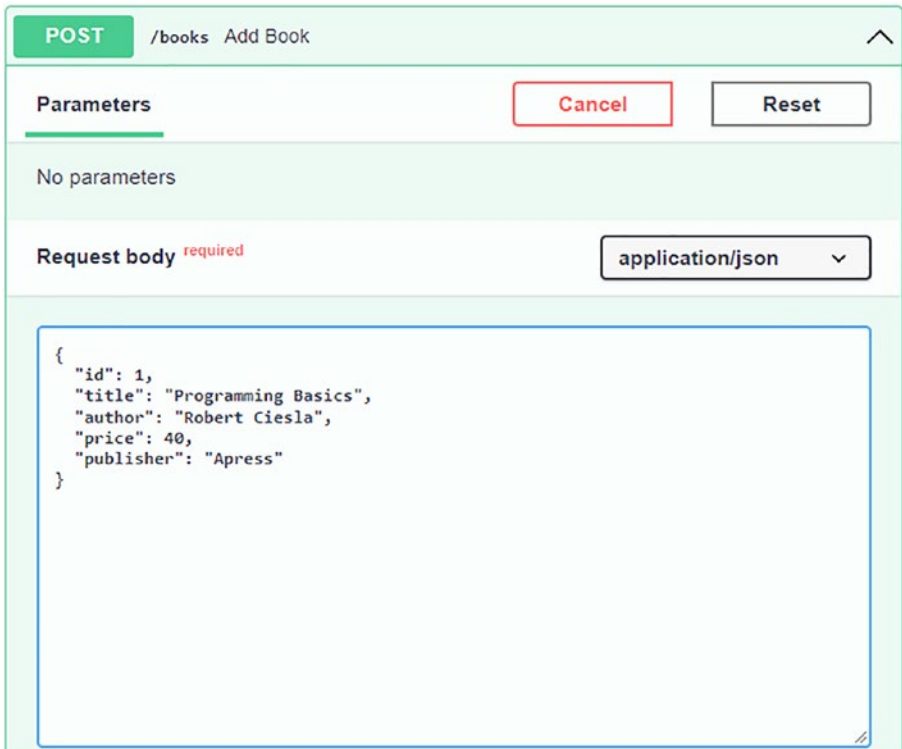
Listing 6-7. POST operation – SQLite

```
from fastapi import FastAPI, Depends, Body

app=FastAPI()

@app.post("/books")
def add_book(book: Book, db=Depends(get_cursor)):
    id=book.id
    title=book.title
    author=book.author
    price=book.price
    publisher=book.publisher
    cur=db[1]
    conn=db[0]
    ins="INSERT INTO books VALUES (?, ?, ?, ?, ?)"
    cur.execute(ins, (id, title, author, price, publisher))
    conn.commit()
    return "Record successfully added"
```

The request body can be populated either by an HTML form or with any HTTP client. Let us test this route with Swagger UI as we have been doing throughout the book. Start the Uvicorn server and expand the `add_book()` function. Enter certain test data and execute the function (Figure 6-1).



POST /books Add Book

Parameters

No parameters

Request body required

application/json

```
{
  "id": 1,
  "title": "Programming Basics",
  "author": "Robert Ciesla",
  "price": 40,
  "publisher": "Apress"
}
```

Figure 6-1. *Swagger UI for POST operation*

Go ahead and insert a few more books. We can check the records in the Books table in the SQLite terminal, as shown in Listing 6-8.

Listing 6-8. Sample Books data

```
sqlite> select * from Books;
```

id	title	author	price	publisher
1	Programming Basics	Robert Ciesla	40	Apress
2	Decoupled Django	Valentino Gag	30	Apress
3	Pro Python	Marty Alchin	37	Apress

Note that the prices are in Euros!

Selecting All Books

As we know, the `@app.get()` decorator mapped to an operation function is employed to read all the records as well as fetch the details of a specific record whose unique ID is passed to it as the path parameter.

First, let us develop a function decorated by `@app.get("/books")`. The `get_books()` function in Listing 6-9 is straightforward. It executes the “**select * from Books;**” query and returns the queryset to the client.

Listing 6-9. GET operation – SQLite: all books

```
@app.get("/books")
def get_books(db=Depends(get_cursor)):
    cur=db[1]
    conn=db[0]

    cur.execute("select * from Books;")
    books=cur.fetchall()
    return books
```

The `http://localhost:8000/books` URL shows the JSON response in the form of a list of dictionary objects (Listing 6-10), each dictionary corresponding to the details of a book.

Listing 6-10. List of books

```
[
  {
    "id": 1,
    "title": "Programming Basics",
    "author": "Robert Ciesla",
    "price": 40,
    "publisher": "Apress"
  },
  {
    "id": 2,
    "title": "Decoupled Django",
    "author": "Valentino Gagliardi",
    "price": 30,
    "publisher": "Apress"
  },
  {
    "id": 3,
    "title": "Pro Python",
    "author": "Marty Alchin",
    "price": 37,
    "publisher": "Apress"
  }
]
```

Selecting a Single Book

The URL path “/books/**1**” implies that the client wants to obtain the details of the book with ID=1. The path parameter is captured in the `@app.get()` decorator and passed to the `get_book()` function, as in Listing 6-11. It also needs the connection context that is made available as the dependency.

Listing 6-11. GET operation – SQLite: single book

```
@app.get("/books/{id}")
def get_book(id: int, db=Depends(get_cursor)):
    cur=db[1]
    conn=db[0]

    cur.execute("select * from Books where id=?", (id,))
    book=cur.fetchone()
    return book
```

The function executes the parameterized SELECT query. The first row in the queryset is obtained by calling the `fetchone()` method of the cursor object and returned to the client as the response. Use `http://localhost:8000/books/1` and the browser displays the first row in the table.

Updating a Book

Listing 6-12 describes the SQL syntax of the UPDATE query.

Listing 6-12. SELECT query syntax

```
UPDATE table_name SET col1=val1, col2=val2,..., colN=valN WHERE
[expression];
```

Let us define the `update_book()` function that internally invokes this query (Listing 6-13). The ID of the book to be updated is passed as the path parameter to the `@app.put()` decorator. The function uses a body parameter to accept the new price of the book.

Listing 6-13. PUT operation – SQLite

```

@app.put("/books/{id}")
def update_book(id:int, price:str=Body(), db=Depends(get_
cursor)):
    cur=db[1]
    conn=db[0]
    qry="UPDATE Books set price=? where id=?"

    cur.execute(qry,(price, id) )
    conn.commit()
    return "Book updated successfully"

```

Issue `http://localhost:8000/books/1` as the URL and submit the new price with the Swagger UI. The record will be updated accordingly. If you run the SELECT query in the SQLite shell, it should show the book with the updated price.

Deleting a Book

Conventionally, the HTTP DELETE method is used to remove the representation of a resource from the server. The URL path pattern for the path decorator is `"/books/{id}"`, as in Listing 6-14. The ID is passed to the `del_book()` function. This function executes the DELETE query.

Listing 6-14. DELETE operation – SQLite

```

@app.delete("/books/{id}")
def del_book(id:int, db=Depends(get_cursor)):
    cur=db[1]
    conn=db[0]

    cur.execute("delete from Books where id=?", (id,))
    conn.commit()
    return "Book deleted successfully"

```


Enter `http://localhost:8000/books/1` in the browser. Check the output of the `SELECT` query in the SQLite shell (or visit the `/books` route) to check the successful removal of the corresponding row from the table.

aiosqlite Module

You may have noted that the operation functions in the preceding example are not coroutines (coroutines are Python functions with the `async` keyword in the beginning of the definition). The reason is that the `sqlite3` module in Python's standard library doesn't support `asyncio`. Its `async-compatible` alternative is the `aiosqlite` module. As it is not part of the standard library, we need to install it with the PIP utility:

```
pip3 install aiosqlite
```

A detailed documentation of the `aiosqlite` module can be found at <https://aiosqlite.omnilib.dev/en/latest/>. The functions and methods in this module have the same nomenclature as that of the `sqlite3` module. However, they are asynchronously executed. As a result, the connection and cursor objects are obtained with the statements in Listing 6-15.

Listing 6-15. Connecting to SQLite with `aiosqlite`

```
conn=await aiosqlite.connect("mydata.sqlite3")
cur=await conn.cursor()
```

The `execute()` method and the methods to fetch rows from the `queryset` are also awaitable (Listing 6-16).

Listing 6-16. Asynchronous fetch methods

```
await conn.execute('SELECT * FROM books')
row = await cur.fetchone()
rows = await cur.fetchall()
```

Let us change all the functions in the example in the previous section so that they can be called asynchronously.

Listing 6-17 gives the **coroutine** version of the `get_cursor()` function, which injects the database context objects into the operation functions.

Listing 6-17. Asynchronous dependency function

```
import aioredis

async def get_cursor():
    conn=await aioredis.connect("mydata.sqlite3")
    conn.row_factory = aioredis.Row
    cur=await conn.cursor()
    yield (conn,cur)
```

Add the `async` keyword to the definition of the POST operation function `add_book()`, as in Listing 6-18. The statements to execute the **INSERT** query and the call to `commit()` have an `await` prefix.

Listing 6-18. POST operation – aioredis

```
@app.post("/books")
async def add_book(book: Book, db=Depends(get_cursor)):
    id=book.id
    title=book.title
    author=book.author
    price=book.price
    publisher=book.publisher
    cur=db[1]
    conn=db[0]
    ins="INSERT INTO books VALUES (?, ?, ?, ?, ?)"
    await cur.execute(ins, (id, title, author, price, publisher))
    await conn.commit()
    return "Record successfully added"
```

Likewise, the coroutine versions of other operation functions – `get_books()`, `update_book()`, and `del_book()` – can be easily constructed. The entire code for the **aiosqlite** version of the application can be found in the book’s code repository.

SQLAlchemy

The CRUD operations in the examples in the previous sections are done on the SQLite database with `sqlite3` and `aiosqlite` modules. For other databases, you’ll use respective DB-API-compatible modules (e.g., `pymysql` for `mysql` and `aiomysql` as its `asyncio`-compatible version). Inside the path operation functions, you basically use the request data to construct the SQL query and then execute it. At times, this can be a tedious task. Various ORMs (**object-relational mappers**) make life easy for the developer.

When you are required to work with a relational database from inside a Python program, you are facing two challenges. Firstly, you must have a good knowledge of the SQL syntax. Secondly, the conversion of data from a Python environment to SQL data types is required. The SQL data types are basically scalar in nature (number, string, etc.), while in Python, you have objects that may comprise more than one primary data type.

The ORM technique helps in converting data between these incompatible type systems. The word **Object** in ORM refers to the object of a Python class. You may recall that, in the theory of relational databases, a table is called a **relation**. A Python class is mapped to a table of corresponding structure in the database. As a result, each object of the mapped class reflects as a row in the database table.

The ORM library enables you to manipulate the object data as per the OO principles. The corresponding SQL statements will be emitted by the ORM, and the CRUD operations will be done behind the scenes. So, you, as a Python developer, don’t have to write a single SQL query!

SQLAlchemy is a very popular SQL toolkit and an object-relational mapper API. In our example, we have been performing CRUD operations on a Books table. Using SQLAlchemy, we'll have a Books class and perform CRUD operations through its object.

Before we proceed, it must be noted that SQLAlchemy internally uses the DB-API driver module of the corresponding type of database. It means SQLAlchemy for a SQLite database uses the `sqlite3` module which is already present. But SQLAlchemy for MySQL needs the `pymysql` or `mysqlclient` module to be installed. The URI to be used for connecting with the database refers to this module. For example, the URI for a SQLite database is **sqlite:///./mydata.sqlite3**, whereas for a MySQL database, it is **mysql+pymysql://root:root@localhost/mydata**. It may be recalled that SQLite is a serverless database, so it doesn't need the user credentials, but for MySQL, the user credentials along with the server's address have to be specified.

A detailed discussion of SQLAlchemy is beyond the scope of this book. The steps involved are briefly explained here:

a. **Connect to a database**

The `create_engine()` establishes the connection with the database. The database is referred to by the **SQLALCHEMY_DATABASE_URL** constant (Listing 6-19). It is essentially similar to the Connection object.

Listing 6-19. SQLAlchemy engine object

```
from sqlalchemy import create_engine
from sqlalchemy.dialects.sqlite import *
SQLALCHEMY_DATABASE_URL = "sqlite:///./mydata.sqlite3"
engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_
    thread": False})
```

b. ORM model

Next, we declare a `Books` class that inherits the `declarative_base` class (Listing 6-20). The class attributes of the `Books` class correspond to the desired structure of a database table mapped to it. The name of the table can be explicitly specified as `__tablename__`.

Listing 6-20. `Books` class as the ORM model

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
from sqlalchemy import Column, Integer, String

class Books(Base):
    __tablename__ = 'book'
    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String(50), unique=True)
    author = Column(String(50))
    price = Column(Integer)
    publisher = Column(String(50))

Base.metadata.create_all(bind=engine)
```

The `create_all()` function creates the tables corresponding to all the classes declared. We have only one, `Books` class, mapped to the `book` table.

c. Session object

A session is similar to the cursor object. It is the handle to the database in use. All the database manipulations are done through this session object. The `Sessionmaker()` function returns in Listing 6-21 the session class.

Listing 6-21. Session object

```
from sqlalchemy.orm import sessionmaker, Session
session = sessionmaker(autocommit=False, autoflush=False,
bind=engine)
```

You may recall that we need to inject the database context into the path operation functions of a FastAPI application. We shall use the function in Listing 6-22 to inject the session object to all the operation functions.

Listing 6-22. Dependency function for SQLAlchemy

```
def get_db():
    db = session()
    try:
        yield db
    finally:
        db.close()
```

d. Pydantic model

The SQLAlchemy part of the application is over with the preceding three steps. For the FastAPI app, we need a Pydantic model class with its structure matching with the ORM class. Listing 6-23 provides the definition of the Book class.

Listing 6-23. Book Pydantic model

```
from pydantic import BaseModel

class Book(BaseModel):
    id: int
    title: str
    author: str
```

```
price:int
publisher: str

class Config:
    orm_mode = True
```

The `orm_mode=True` setting in the `BaseModel` configuration allows you to convert a Pydantic model object to a SQLAlchemy object and vice versa.

Now, we shall develop the path operation functions for POST, GET, PUT, and DELETE methods. We shall use the same URL path endpoints as we have done in earlier examples:

- POST /books
- GET /books
- GET /books/{id}
- PUT /books/{id}
- DELETE /books/{id}

e. **@app.post()**

You know that the INSERT operation is performed by the HTTP POST method. Use the `Book` Pydantic model as the `response_model` for this purpose and its object as the response body parameter for the `add_book()` function, as in Listing 6-24.

Listing 6-24. POST operation – SQLAlchemy

```

from fastapi import FastAPI, Depends
from typing import List

app=FastAPI()

@app.post('/books', response_model=Book)
def add_book(b1: Book, db: Session = Depends(get_db)):
    bkORM=Books(**b1.dict())
    db.add(bkORM)
    db.commit()
    db.refresh(bkORM)
    return "Book added successfully"

```

The Pydantic model object can be directly unpacked into the SQLAlchemy object, whose `add()` method inserts a new row in the database table. Use the same data to insert records (Listing 6-8).

f. **@app.get()**

This decorator is associated with the `get_books()` function. With the help of the database session object, query the model and retrieve all the currently available objects. The collection is returned as a list of Book – the Pydantic BaseModel.

Listing 6-25 also has the code for the `get_book()` function. The path parameter in the URL parses the ID of the object to be retrieved. The function fetches the object with the given ID from the Books model and returns it as the response.

Listing 6-25. GET operation – SQLAlchemy

```

@app.get('/books', response_model=List[Book])
def get_books(db: Session = Depends(get_db)):
    recs = db.query(Books).all()
    return recs

@app.get('/books/{id}', response_model=Book)
def get_book(id:int, db: Session = Depends(get_db)):
    return db.query(Books).filter(Books.id == id).first()

```

g. @app.put()

To perform the update operation, we pass the ID of the object to be updated as the path parameter and the new price as the body parameter. The `update_book()` function in Listing 6-26 fetches the object as done in the preceding step and updates the price, simply by assigning it the new value from the request body.

Listing 6-26. PUT operation – SQLAlchemy

```

@app.put('/books/{id}', response_model=Book)
def update_book(id:int, price:int=Body(), db: Session =
Depends(get_db)):
    bkORM = db.query(Books).filter(Books.id == id).first()
    bkORM.price=price
    db.commit()
    return bkORM

```

h. `@app.delete()`

The DELETE operation is fairly straightforward (Listing 6-27).

The `del_book()` function receives the ID from the URL and runs the `delete()` method on the retrieved object.

Listing 6-27. GET operation – SQLAlchemy

```
@app.delete('/books/{id}')
def del_book(id:int, db: Session = Depends(get_db)):
    try:
        db.query(Books).filter(Books.id == id).delete()
        db.commit()
    except Exception as e:
        raise Exception(e)
    return "book deleted successfully"
```

You can obtain the complete code for the example explained in this section from the code repository and try it out.

async in SQLAlchemy

The SQLAlchemy ORM doesn't yet completely support asynchronous operations. The latest version of SQLAlchemy (ver. 1.4.40) does have this feature, albeit on an experimental basis. Furthermore, it hasn't been extended to all types of databases.

However, the other branch of SQLAlchemy – called **SQLAlchemy Core** – can be used for performing asynchronous database operations with the help of the `databases` module.

The SQLAlchemy Core package uses a schema-centric SQL **Expression Language**. In this section, we shall briefly explore how the SQL queries are handled with the Expression Language.

databases Module

The databases module differs a little from the DB-API standard. It provides functions for connecting to a database, executing queries, and fetching table data. These functions support `async/await`. We need to install the databases module with the PIP installer:

```
pip3 install databases
```

The databases module provides `asyncio` support for PostgreSQL, MySQL, and SQLite. It uses the SQLAlchemy Core Expression Language to construct SQL queries.

You also need to install the `asyncio`-compatible drivers – `asyncpg` and `aiopg` for PostgreSQL, `aiomysql` and `asyncmy` for MySQL, and `aiosqlite` for SQLite databases. We have already used the `aiosqlite` module earlier in this chapter.

To establish a connection with a SQLite database, import the databases module, set the Database object, and call its `connect()` method from inside a coroutine (Listing 6-28).

Listing 6-28. Connecting using the databases module

```
import databases

DATABASE_URL = "sqlite:///./mydata.sqlite3"

db = databases.Database(DATABASE_URL)

async def connection():
    await db.connect()
```

Next, how do we create a table in this database? With SQLAlchemy's Expression Language, we can represent relational database structures and expressions using Python code.

Core Expression Language

The Engine class provides a source of database connectivity and behavior. An object of the Engine class is instantiated using the `create_engine()` function (Listing 6-29).

Listing 6-29. Engine object

```
import sqlalchemy

DATABASE_URL = "sqlite:///./mydata.sqlite3"

engine = sqlalchemy.create_engine(DATABASE_URL,
connect_args={"check_same_thread": False})
```

In SQLAlchemy Core, an object of the Metadata class is a collection of Table objects and their associated schema constructs (Listing 6-30).

Listing 6-30. MetaData object

```
metadata = sqlalchemy.MetaData()
```

Add the booklist table in the metadata object with the code in Listing 6-31.

Listing 6-31. Table object

```
booklist = sqlalchemy.Table(
    "booklist",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_
key=True),
    sqlalchemy.Column("title", sqlalchemy.String),
    sqlalchemy.Column("author", sqlalchemy.String),
    sqlalchemy.Column("price", sqlalchemy.Integer),
    sqlalchemy.Column("publisher", sqlalchemy.String),
)
```

```
metadata.create_all(engine)
```

The `create_all()` method creates all the tables available in the `metadata`.

As we have done in previous sections, define a coroutine that injects the Database object into the path operation functions. Listing 6-32 defines `get_db()` coroutine.

Listing 6-32. Asynchronous dependency function

```
async def get_db():
    db = databases.Database(DATABASE_URL)
    await db.connect()
    yield db
```

Table Class Methods

Before we try to develop the FastAPI app and its path operations, we need to know how the CRUD operations are performed with the Core Expression Language.

The `Table` class in the `sqlalchemy` module has the corresponding methods for performing insert, select, update, and delete operations.

The `insert()` method returns a string representation of the SQL **INSERT** query which in turn is run with the `execute()` function (Listing 6-33).

Listing 6-33. Insert method of the `Table` class

```
query=table.insert().values(field1=value1, field2=value2, ...)
db.execute(query)
```

The `select()` method of the `Table` class constructs the **SELECT** query. To retrieve one or all the records in the queryset, use the `fetch_one()` or `fetch_all()` function as per the syntax in Listing 6-34.

Listing 6-34. Fetch methods of the Table class

```
query=table.select().where(condition)
rows=db.fetch_all(query)
row=db.fetch_one(query)
```

The `update()` method internally emits the **UPDATE** query as in SQL (Listing 6-35).

Listing 6-35. Update method of the Table class

```
query=table.update().where(condition).values(field1,value1, ..)
db.execute(query)
```

Lastly, the `delete()` method helps in forming the **DELETE** query (Listing 6-36).

Listing 6-36. Delete method of the Table class

```
query=table.delete().where(condition)
db.execute(query)
```

This gives a general syntax of how the database table operations are done. We shall see these functions in action when we develop the path operations of our FastAPI app.

FastAPI Path Operations

The rest of the steps are the same as we did in the previous sections. Our Pydantic model remains the same. Let us start with the POST operation. The `add_book()` function is now a coroutine. (Note that we shall be using the same URL endpoints and the operation functions as in the previous sections.)

As in the earlier sections, the post operation function (a coroutine really) uses the response body parameters populated by the Pydantic model object and uses its attributes as the values of the new row to be inserted. Listing 6-37 contains the code for the `add_book()` coroutine.

Listing 6-37. POST operation – SQLAlchemy Core

```
@app.post("/books", response_model=Book)
async def add_book(b1: Book, db=Depends(get_db)):
    query = booklist.insert().values(id=b1.id, title=b1.title,
    author=b1.author, price=b1.price, publisher=b1.publisher)
    await db.execute(query)
    return "Book added successfully"
```

Note that the call to the `execute()` is asynchronous, as it is prefixed by the `await` keyword.

Insert the data in the table as per the previous sections.

The `get_books()` coroutine (Listing 6-38) is invoked when a GET request for the **/books** URL is received. It simply returns the queryset obtained with the `select()` method.

Listing 6-38. GET operation – SQLAlchemy Core – all books

```
@app.get("/books", response_model=List[Book])
async def get_books(db=Depends(get_db)):
    query = booklist.select()
    return await db.fetch_all(query)
```

To retrieve a specific record from the table, the value of its **id** (it is the table's primary key) should be passed as the path parameter. Inside the `get_book()` coroutine (in Listing 6-39), the **id** becomes the condition for the where clause when the `select()` method is called. It fetches the corresponding row, which is returned as the response.

Listing 6-39. GET operation – SQLAlchemy Core – single book

```
@app.get("/books/{id}")
async def get_book(id: int, db=Depends(get_db)):
    query=booklist.select().where(booklist.c.id==id)
    return await db.fetch_one(query)
```

The PUT operation is handled by the `update_book()` coroutine, as in Listing 6-40. As we intend to update the price of a specific record, its `id` is included in the URL as the path parameter and the new price as the body parameter.

Listing 6-40. PUT operation – SQLAlchemy Core

```
@app.put("/books/{id}")
async def update_book(id:int, new_price:int=Body(),
db=Depends(get_db)):
    query=booklist.update().where(booklist.c.id==id).
    values(price=new_price)
    await db.execute(query)
    return "Book updated successfully"
```

The `update()` method uses the path parameter – **id** – to apply the filter in the **where** clause.

Finally, the `del_book()` coroutine is mapped to the **DELETE** operation decorator (Listing 6-41). It parses the primary key from the path and invokes the `delete()` method of the table class as in Listing 6-41.

Listing 6-41. DELETE operation – SQLAlchemy Core

```
@app.delete("/books/{id}")
async def del_book(id:int, db=Depends(get_db)):
    query=booklist.delete().where(booklist.c.id==id)
```


In this way, the asynchronous handling of the path operations is done in a FastAPI application over a SQLite database, with the help of the SQLAlchemy Core Expression Language.

All the code segments in this section are put together in a complete Python source code, which is present in the book's repository.

PyMongo for MongoDB

So far in this chapter, we learned how to use a relational database (with SQLite as an example) as a back end to a FastAPI app. In this section, we shall learn to use a MongoDB database.

MongoDB is a schemaless, document-oriented, **NoSQL** database. It stores semistructured documents in **BSON** format – a binary serialization of JSON-like documents. A Document in a MongoDB database is a collection of key-value pairs – similar to a Python dictionary object. One or more such documents are stored in a Collection. For analogy, you can think of a Collection in MongoDB as equivalent to a table in a relational database, and a Document is similar to a single row in a table of a SQL-based relational database.

The MongoDB community edition is available for download and installation at www.mongodb.com/download-center/community. Assuming that MongoDB is installed on Windows in the e:\mongodb folder, start the MongoDB server from the command terminal using the command in Listing 6-42.

Listing 6-42. Starting the MongoDB server

```
E:\mongodb\bin>mongod
..
waiting for connections on port 27017
```

You can now interact with the MongoDB server by launching the MongoDB shell with the Mongo command in another terminal:

```
E:\mongodb\bin>mongo
MongoDB shell version v4.0.6
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("eda5ab88-8ee3-49dd-8469-1545650f68b1") }
MongoDB server version: 4.0.6
Welcome to the MongoDB shell.
For interactive help, type "help".
..
..
>
```

The MongoDB Query Language is based on JavaScript. Database, Collection, and Documents can be added using the MongoDB queries.

Instead of using the query syntax, you can also use a GUI tool called MongoDB **Compass**. To start using it, first establish a connection with the MongoDB server already running. Figure 6-2 shows the New Connection dialog screen where you need to provide **localhost** as the hostname and **27017** as the port number.

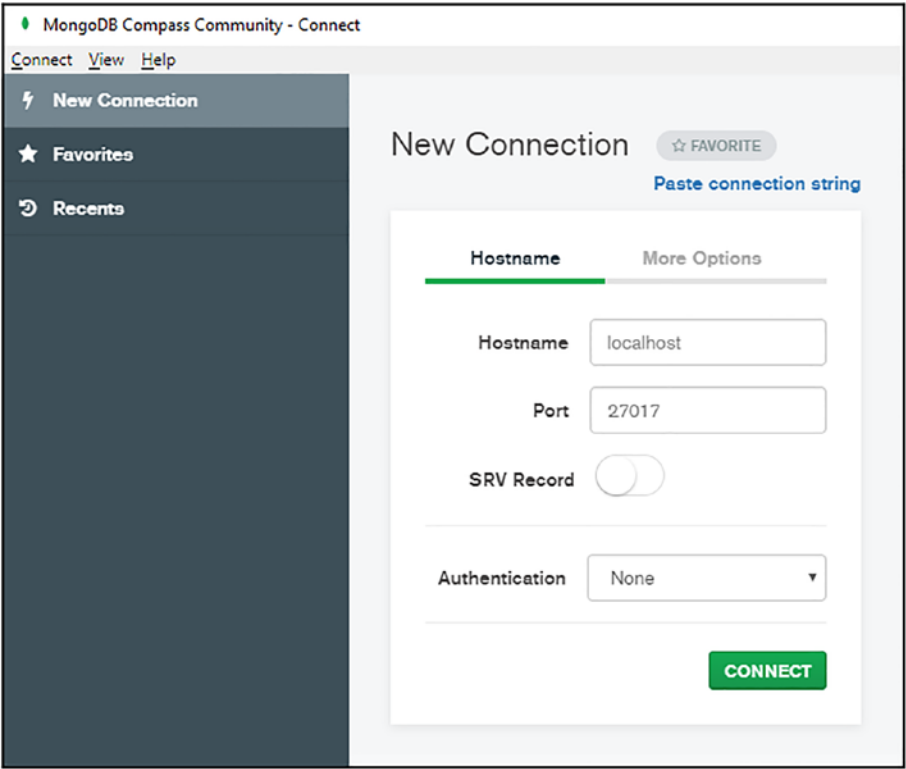


Figure 6-2. MongoDB Compass

You will get to see the list of databases on the server once the connection is successfully established (Figure 6-3).

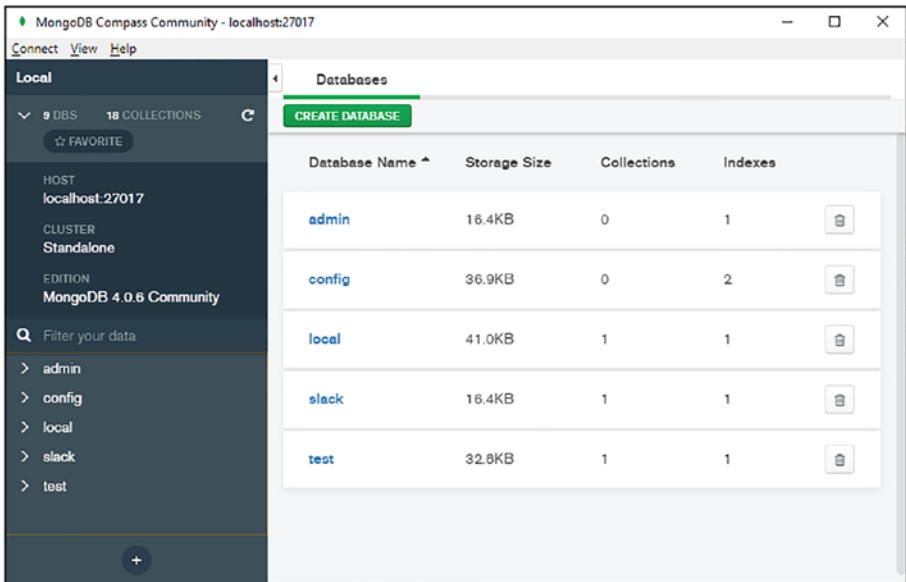


Figure 6-3. Databases on the MongoDB server as shown in Compass

For relational databases, we need to use DB-API-compliant driver modules. For SQLite, we used `sqlite3` and `aiosqlite` modules for asynchronous access. Similarly, we need to use an interface between the Python application and MongoDB. PyMongo is the official Python driver for MongoDB. It can be easily installed with the PIP utility:

```
pip3 install pymongo
```

An object of the `MongoClient` class in this module establishes the connection with the MongoDB server. You can create a new database and a collection in it with the function in Listing 6-43.

Listing 6-43. Function to inject Collection object context

```
def get_collection():
    client=MongoClient()
    DB = "mydata"
    coll = "books"
    bc=client[DB][coll]
    yield bc
```

We are, in fact, going to use this function to inject the Collection object as the dependency for the application routes in the FastAPI app.

We shall be using the same Pydantic model, the same URL endpoints, and the same path operation functions that we have used in all the preceding sections.

The `add_book()` function (in Listing 6-44) reads the Pydantic model object from the request body. The `insert_one()` method of the PyMongo Collection object adds a new document to the collection. The dict representation of the Pydantic object goes as an argument for the `insert_one()` method.

Listing 6-44. POST operation – PyMongo

```
@app.post("/books")
def add_book(b1: Book, bc=Depends(get_collection)):
    result = bc.insert_one(b1.dict())
    return "Book added successfully"
```

Perform the POST request operation to add three documents. (Please refer to Listing 6-8.) The Compass GUI shows the JSON representation of these documents as in Listing 6-45.

Listing 6-45. BSON representation of MongoDB Documents

```

{
  "_id": {
    "$oid": "63734d09d43843bf5a32f0ba"
  },
  "bookID": {
    "$numberInt": "1"
  },
  "title": "Programming Basics",
  "author": "Robert Ciesla",
  "price": {
    "$numberInt": "40"
  },
  "publisher": "Apress"
}

{
  "_id": {
    "$oid": "6373dd6376dbe609ce2a365c"
  },
  "bookID": {
    "$numberInt": "2"
  },
  "title": "Decoupled Django",
  "author": "Valentino Gag",
  "price": {
    "$numberInt": "30"
  },
  "publisher": "Apress"
}

```

```
{
  "_id": {
    "$oid": "6373e3d49f5ae7216c810ca2"
  },
  "bookID": {"$numberInt": "3"},
  "title": "Pro Python",
  "author": "Marty Alchin",
  "price": {...},
  "publisher": "Apress"
}
```

Note that MongoDB automatically generates “**_id**” as the primary key of the document.

The **GET /books** request is handled by the `get_books()` function (Listing 6-46). It essentially returns the list of documents available in the collection, fetched by the `find()` method.

Listing 6-46. GET operation – PyMongo – all books

```
@app.get("/books", response_model=List[Book])
def get_books(bc=Depends(get_collection)):
    books=list(bc.find())
    return books
```

To retrieve a single document, we pass its BookID in the URL. The `@app.get()` decorator captures it as a path parameter. Inside the `get_book()` function, we retrieve the document by calling the `find_one()` method (Listing 6-47), applying the path parameter as the filter.

Listing 6-47. GET operation – PyMongo – single book

```
@app.get("/books/{id}", response_model=Book)
def get_book(id: int, bc=Depends(get_collection)):
    """Get all messages for the specified channel."""
    b1=bc.find_one({"bookID": id})
    return b1
```

In this way, the path operations are implemented in a FastAPI application.

Motor for MongoDB

The PyMongo driver doesn't support `async/await`. MongoDB itself provides an asynchronous Python driver for MongoDB – called **Motor**. It is a coroutine-based API for nonblocking access to MongoDB from `asyncio`.

To start with, install Motor in the current Python environment with PIP:

```
pip3 install motor
```

The Motor package itself is structured on top of PyMongo. It can be said that Motor is the asynchronous version of PyMongo. Hence, most of the PyMongo API is refactored to handle the `async/await` feature of modern Python.

The client class in Motor is now `AsyncIOMotorClient` in place of `MongoClient` as in PyMongo. Motor's client class does not connect to the database as soon as it is instantiated (as in the case of PyMongo). Instead, it connects on demand, when you first attempt an operation.

Hence, in Listing 6-48, we change our `get_collection()` function that injects the collection object into the operation functions.

Listing 6-48. Connecting with Motor

```
import motor.motor_asyncio

def get_collection():
    client=motor.motor_asyncio.AsyncIOMotorClient()
    DB = "mydb"
    coll = "books"
    bc=client[DB][coll]
    yield bc
```

We need to effect two changes to the path operation functions. First, make them coroutines by affixing the `async` keyword, and, second, add `await` to the CRUD methods of the collection object.

Once again, we use the same URL path endpoints and the same path operation functions (now they become coroutines) as we have been using in the earlier sections.

The POST operation decorator and its path operation coroutine are now as shown in Listing 6-49.

Listing 6-49. POST operation – Motor

```
@app.post("/books")
async def add_book(b1: Book, bc=Depends(get_collection)):
    result = await bc.insert_one(b1.dict())
    return "Book added successfully"
```

Similarly, the `asyncio`-compatible version of `get_books()` is given in Listing 6-50:

Listing 6-50. GET operation – Motor

```
@app.get("/books", response_model=List[Book])
async def get_books(bc=Depends(get_collection)):
    books=await bc.find().to_list(1000)
    return books
```

It is worth noting here that the `queryset` returned by the `find()` method is synchronous, but `add_to_list()` to it so that the `await` keyword can be applied.

To retrieve a single book of the given ID, we write the `get_book()` function as in Listing 6-51.

Listing 6-51. GET operation – Motor – single book

```
@app.get("/books/{id}", response_model=Book)
async def get_book(id: int, bc=Depends(get_collection)):
    b1=await bc.find_one({"bookID": id})
    return b1
```

You can see that using Motor in place of PyMongo doesn't need many changes in the code.

Summary

With this, we conclude an important discussion on how to use relational databases and MongoDB as a back end to a FastAPI app. The classical way of interacting with the relational databases is through DB-API drivers. For asynchronous operations on an SQLite database, we learned how to use the `aiosqlite` module.

We also learned how FastAPI handles a database with SQLAlchemy ORM as well as SQLAlchemy Core (with the help of the `databases` module) and interacts with Pydantic models.

In the end, we discussed the use of MongoDB in a FastAPI app, both with PyMongo and Motor drivers.

CHAPTER 7

Bigger Applications

So far in this book, we have seen that the FastAPI web app is contained in a single Python script (conventionally `main.py`). All the path operation routes, their respective operation functions, the models, all the required imports, etc., are put in the same code file.

Things will become messier if the application involves handling of more than one resources. Take the case of an ecommerce app where CRUD operations on books as well as music albums are to be performed. Putting all the HTTP operations for both the products along with their models is not the ideal way to organize the application code.

In this chapter, we shall learn how to deal with the challenges of handling bigger applications with more than one API. This chapter explains the following topics:

- Single file app
- APIRouter
- Mounting applications
- Dependencies
- Middleware
- CORS

Single File App

To begin, let us start by defining all the path operations for books and albums in a single script. The structure of the app would appear as in Listing 7-1.

Listing 7-1. App with two APIs

```
from fastapi import FastAPI
from pydantic import BaseModel

class book(BaseModel):
    id:int

class album(BaseModel):
    id:int

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Home page"}

#routes for books API
@app.get("/books")
async def get_books():
    return "pass"

@app.get("/books/{id}")
async def get_book(id:int):
    return "pass"

@app.post("/books")
async def add_book(b1:book):
    return "pass"

@app.put("/books/{id}")
```

```

async def update_book(id:int):
    return "pass"
@app.delete("books/{id}")
async def del_book(id:int):
    return "pass"

#routes for album API
@app.get("/albums")
async def get_album():
    return "pass"
@app.get("/albums/{id}")
async def get_album(id:int):
    return "pass"
@app.post("/albums")
async def add_album(a1:album):
    return "pass"
@app.put("/albums/{id}")
async def update_album(id:int):
    return "pass"
@app.delete("albums/{id}")
async def del_album(id:int):
    return "pass"

```

You can imagine how lengthy this code would become as you go on expanding the models with their field structure and functions with their respective processing logic. Obviously, it will be difficult to debug and maintain such a lengthy code.

Even the Swagger UI docs page looks very ungainly (Figure 7-1).



Figure 7-1. *Swagger UI for two APIs in a single app*

Clearly, we need to organize the code in a more structured manner. The `APIRouter` class in FastAPI is the tool for this purpose.

APIRouter

The use of `APIRouter` allows you to group your routes into different file structures so that they are easily manageable. All the routes in a bigger application, such as the preceding example, are clubbed into small units of `APIRouters`. The individual `APIRouters` are then included in the main application.

Note that these smaller units are not independent applications. They can be considered to be mini FastAPI apps that are part of the bigger application. All the routes from all the `APIRouters` will become a part of the main application documentation.

Let us implement this concept and rearrange the code in the preceding application. The routes related to the book resource and the book model are stored in the `books.py` file. Similarly, the album routes and album model go in the `albums.py` script.

First, declare an object of the `APIRouter` class (it is in the `fastapi` module) instead of the `FastAPI` class as we normally do (Listing 7-2). Set the `prefix` attribute to `"/books"` and define a tag that appears in the documentation.

Listing 7-2. `APIRouter` class

```
from fastapi import APIRouter

books = APIRouter(prefix="/books",
                  tags=["books"])
```

As far as the **REST** operations on the book resource are concerned, `books` - the `APIRouter` object itself is the application object. Hence, the path operation decorators are `@books.get()`, `@books.post()`, etc.

Save the script (Listing 7-3) as **books.py**.

Listing 7-3. books router

```
from fastapi import APIRouter
from pydantic import BaseModel

books = APIRouter(prefix="/books",
                  tags=["books"])

class book(BaseModel):
    id:int

#routes for books API
@books.get("/")
async def get_books():
    return "pass"
@books.get("/{id}")
async def get_book(id:int):
    return "pass"
@books.post("/")
async def add_book(b1:book):
    return "pass"
@books.put("/{id}")
async def update_book(id:int):
    return "pass"
@books.delete("/{id}")
async def del_book(id:int):
    return "pass"
```

Note that book – the Pydantic model – is also present in the code.
The **albums.py** script (Listing 7-4) is along the similar lines.

Listing 7-4. albums router

```

from fastapi import APIRouter
from pydantic import BaseModel

albums = APIRouter(prefix="/albums",
                    tags=["albums"])

class album(BaseModel):
    id:int

#routes for albums API
@albums.get("/")
async def get_albums():
    return "pass"
@albums.get("/{id}")
async def get_album(id:int):
    return "pass"
@albums.post("/")
async def add_album(a1:album):
    return "pass"
@albums.put("/{id}")
async def update_album(id:int):
    return "pass"
@albums.delete("/{id}")
async def del_album(id:int):
    return "pass"

```

Now we come to the main FastAPI application code. The application object is declared here. To include the mini applications, import the books and albums modules. Use the APIRouter objects in these modules as the argument to the `include_router()` method of the app object.

Listing 7-5 shows how the **main.py** script should be.

Listing 7-5. Main app with routers

```
from fastapi import FastAPI
import books, albums

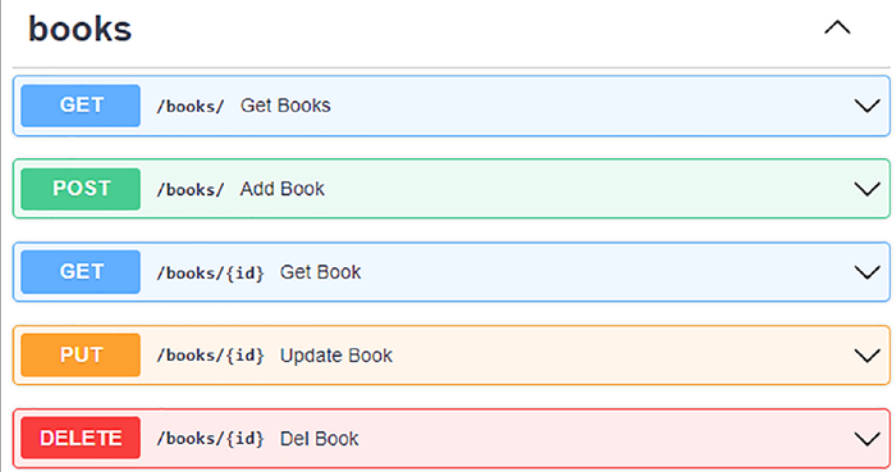
app = FastAPI()

app.include_router(books.books)
app.include_router(albums.albums)

@app.get("/")
async def root():
    return {"message": "Home page"}
```

Run the server and open the Swagger documentation page. The path operation functions are nicely grouped together with the prefix tag defined in each APIRouter declaration.

The **books** API routes are as in Figure 7-2.



books			
GET	/books/	Get Books	▼
POST	/books/	Add Book	▼
GET	/books/{id}	Get Book	▼
PUT	/books/{id}	Update Book	▼
DELETE	/books/{id}	Del Book	▼

Figure 7-2. Routes of the *books* subapp

On the other hand, the **albums** routes appear in another group, as in Figure 7-3.

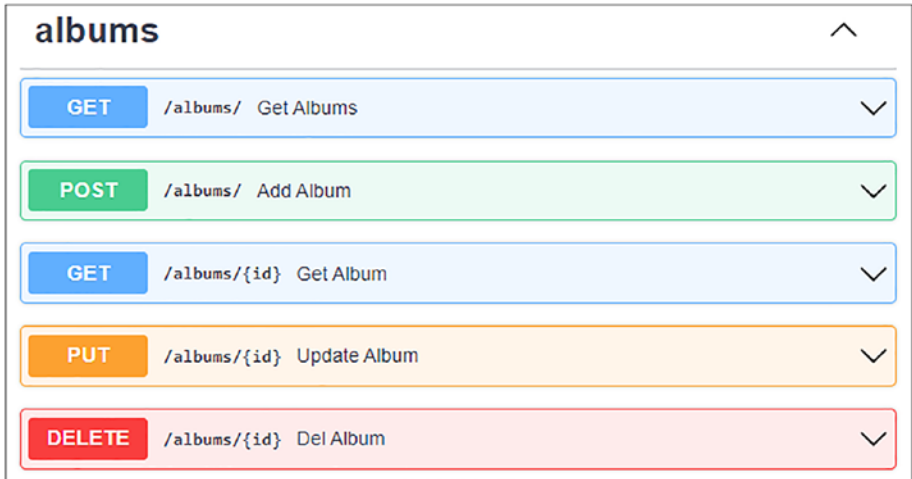


Figure 7-3. Routes of the albums subapp

Router Package

To further refine the app structure, create two folders inside the application folder. Name them `albums` and `books`. Move the `APIRouter` code files `albums.py` and `books.py` in the `albums` and `books` folders.

Place an empty `__init__.py` file in both the subfolders so that Python recognizes them as packages.

Remove the model declarations from the router code and put the same in `models.py` in its respective folder. The file structure should be as shown in Figure 7-4.



Figure 7-4. File structure for APIRouter packages

The code in the **albums/albums.py** file is changed to the one in Listing 7-6.

Listing 7-6. albums module

```

from fastapi import APIRouter
from .models import album

albums = APIRouter(prefix="/albums",
                   tags=["albums"])

#routes for albums API
#keep the operation functions as it is

```

Note that the album Pydantic model has been moved to the **albums/models.py** file, as in Listing 7-7.

Listing 7-7. album model in the albums package

```

from pydantic import BaseModel

class album(BaseModel):
    id:int

```

The books package contents follow the same approach.

The script for **main.py** (Listing 7-8) changes a bit, as the `APIRouter` objects need to be imported from the packages.

Listing 7-8. Main app code

```
from fastapi import FastAPI
from books import books
from albums import albums

app = FastAPI()

app.include_router(books.books)
app.include_router(albums.albums)

@app.get("/")
async def root():
    return {"message": "Home page"}
```

Mounting Subapplications

The `APIRouter` class helps you to include a subapplication to the main `FastAPI` object. If, on the other hand, you have two (or more) apps that are independent of each other, having their own API and docs, you can designate one of them as the main app and mount others.

Earlier in the book (Chapter 4, section “Serving Static Assets”), we have used the `mount()` function to make the static assets available at the “/static” route. The `StaticFiles` class is really a subapplication. We mount this subapp on the main application object, so that the static files are available for use, especially in the templates.

Let us extend this concept to mount the albums and books objects to the main app. We shall keep the same file and folder structure, where the albums and books packages are present in the main app folder.

Inside the **albums/albums.py** module (Listing 7-9), change the books object to a regular FastAPI object instead of the APIRouter object.

Listing 7-9. albums subapp

```
from fastapi import FastAPI

albums=FastAPI()

#routes for albums API
@albums.get("/albums")
async def get_albums():
    return "pass"
#define rest of the path operation functions here
```

Do the same with the **books/books.py** file (Listing 7-10).

Listing 7-10. books subapp

```
from fastapi import FastAPI

books=FastAPI()

#routes for books API
@books.get("/books")
async def get_books():
    return "pass"
#define rest of the path operation functions here
```

Note that both these apps are stand-alone and can be run independently.

With the OS command terminal inside the albums folder, run the following command to start the albums app:

```
uvicorn albums:albums --reload
```

The `http://localhost:8000/docs` URL lets you inspect the docs of the `albums` app.

Similarly, run the other independent app – `books` – from inside the `books` folder:

```
uvicorn books:books --reload
```

The `http://localhost:8000/docs` URL now shows the Swagger UI of the `books` app.

To mount these apps as subapps inside the main application code, import the `albums` and `books` objects from their respective packages and mount them on the main app object (Listing 7-11).

Listing 7-11. Mounting subapps

```
from fastapi import FastAPI
from albums import albums
from books import books

app = FastAPI()

@app.get("/stores")
async def root():
    return {"message": "Home page"}

app.mount("/albumapi", albums.albums)
app.mount("/bookapi", books.books)
```

After running the server, the `http://localhost:8000/docs` URL displays the routes of only the main app. The routes of the `albums` app are now mounted on “**/albumapi**”; hence, the routes of this subapp (Figure 7-5) are now available at `http://localhost:8000/albumapi/docs`.

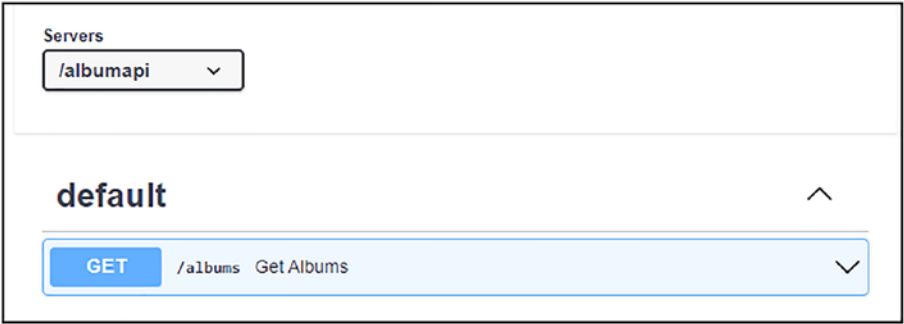


Figure 7-5. Routes of the mounted albums app

Similarly, the documentation of the books app is now found (Figure 7-6) at <http://localhost:8000/bookapi/docs>.

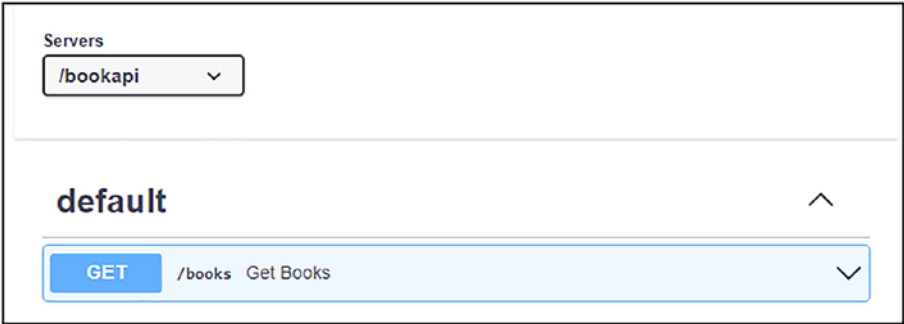


Figure 7-6. Routes of the mounted books subapp

Dependencies

A path operation function in a FastAPI app may need parameters in its context for processing. The user passes path and query parameters in the request URL itself. On the other hand, the body parameters are read from the request body, usually POSTed by an HTML form.

Sometimes, though, the path operation function depends upon certain other factors. These are called the dependencies of the operation. We need to inject these dependencies into the function's context. FastAPI has a `Depends()` function whose return value is injected as the dependency parameter.

You may recall that we have used this `Depends()` parameter in the previous chapter (**Chapter 6 – Using Databases**). Obviously, each path function that performs CRUD operations on a database needs the connection object. Instead of setting the connection object locally inside each function, we used the `get_db()` function and injected its return value in each operation function.

The mechanism of dependency injection is useful especially when the same code logic needs to be used across multiple path operations. It also helps in enforcing authentication checks and ensuring the requirements, such as allowing access to users with specific roles, etc.

Example of Dependency Injection

Let us try to understand how dependency in FastAPI works, with the help of this simple example. Let us assume that the services of a web application are not available to the users every Sunday. Naturally, when a user hits any URL, its mapped function will be required to find out whether it's Sunday. Instead of putting the code for this purpose in every path operation function, a more convenient approach is to write a function and use it to inject the dependency.

Listing 7-12 shows a simple Python function that returns a number corresponding to the weekday of the current date. For Monday, it returns 0; for Sunday, it returns 6.

Listing 7-12. Dependency function

```
def dow():
    from datetime import datetime
    dow=datetime.now().weekday()
    return dow
```

Let us now use this function as an argument for Depends in the path operation coroutine (Listing 7-13).

Listing 7-13. Using Depends()

```
@app.get("/")
async def root(day=Depends(dow)):
    if day==6:
        return {"message": "Service not available on Sunday"}
    return {"message": "Home page"}
```

If the user enters the “/” route on Sunday, they get the “Service not available” message; on other days, the JSON response of the app is a “Home Page” message.

Query Parameters As Dependencies

The dependency function can be a normal function or a coroutine defined with `async def`. An async dependency can be called from within an async path operation function or a normal path operation function as well. In the same way, a normal dependency can also be called from async as well as normal path operation function.

In the example (Listing 7-14), the FastAPI application has two routes that fetch the list of persons and employees, respectively. The range of the list is determined by a dependency function `properties()`, which is defined with the `async` keyword.

Listing 7-14. Dependency function with parameters

```

async def properties(x: int, y: int):
    return {"from": x, "to": y}

```

The dictionary object returned by this coroutine injects the query parameters into the `get_persons()` operation function (Listing 7-15).

Listing 7-15. Parameterized dependency injection

```

from fastapi import Depends, FastAPI

app = FastAPI()

persons=[
    {"name": "Tom", "age": 20},
    {"name": "Mark", "age": 25},
    {"name": "Pam", "age": 27}
]

@app.get("/persons/")
async def get_persons(params: dict = Depends(properties)):
    return persons[params['from']:params['to']]

```

Upon receiving the client request for the “/**persons**/” URL path, FastAPI solves the dependency to provide `x` and `y` as query parameters. Check the documentation of the `get_persons()` function in Figure 7-7.

GET

/persons/ Get Persons

^

Parameters

Cancel

Name	Description
<div><div>x * required</div><div>integer</div><div>(query)</div></div>	<div>1</div>
<div><div>y * required</div><div>integer</div><div>(query)</div></div>	<div>3</div>

Figure 7-7. Query parameters as dependencies

FastAPI returns a sliced list of persons[x:y]. Assuming that we provide the values x=1 and y=3, the request URL constructed by the Swagger UI will be `http://localhost:8000/persons/?x=1&y=3` (Figure 7-8).

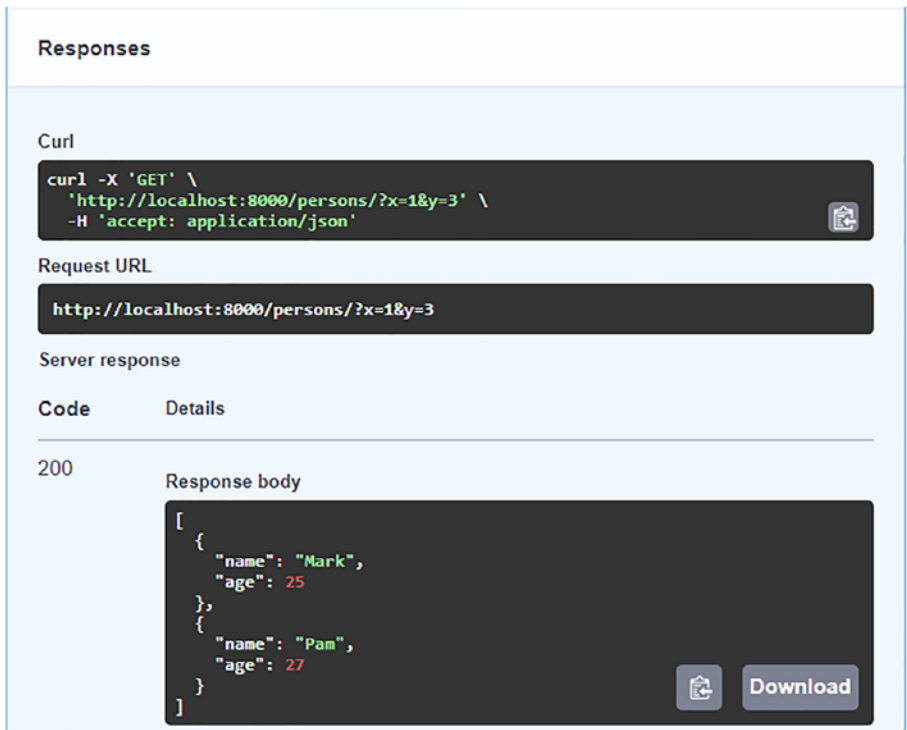


Figure 7-8. Swagger UI with injected query parameters

The same dependency is also utilized to inject the list indices into the `get_employees()` function, as shown in Listing 7-16.

Listing 7-16. `Depends()` injecting query parameters

```
employees=[
    {"name": "Tom", "salary": 20000},
    {"name": "Mark", "salary": 25000},
    {"name": "Pam", "salary": 27000}
]
@app.get("/employees/")
async def get_employees(params: dict = Depends(properties)):
    return employees[params['from']:params['to']]
```

This, in fact, demonstrates the primary purpose of using a dependency function, as the same code logic is used for both the path operations.

Parameterized Dependency Function

The dependency function, being just like any other normal Python function, may have one or more parameters in its declaration. You may even have a dependency function that itself depends on another function.

APIs are normally distributed to the user with an authorization key. At the time of logging in, the user provides the key along with the identity credentials such as ID and password. In the example (Listing 7-17), the “/” route adds these values to the server’s response as the cookies.

Listing 7-17. Path operation to insert cookies

```
from fastapi import FastAPI, Request, Depends
from fastapi.responses import Response

app = FastAPI()

@app.get("/")
async def index(request: Request, response: Response):
    response.set_cookie(key="user", value="admin")
    response.set_cookie(key="api_key", value="abcdef12345")
    return {"message": "Home Page"}
```

When the client requests a GET operation to fetch the list of objects, the application verifies if the API key has been set or not. We define a `credentials()` function in Listing 7-18 for this purpose.

Listing 7-18. Parameterized dependency function

```
def credentials(request):
    dct=request.cookies
    try:
        return dct['api_key']
    except:
        return None
```

Since this verification is necessary for every operation in the application, the `credentials()` function becomes the dependency. Note that this function requires the `Request` object as a parameter to fetch the **'api_key'** cookie. This parameter is passed to it by the path operation function, as in Listing 7-19.

Listing 7-19. Cookies injected by `Depends()`

```
persons=[
    {"name": "Tom", "age": 20},
    {"name": "Mark", "age": 25},
    {"name": "Pam", "age": 27}
]
@app.get("/persons/")
async def get_persons(key=Depends(credentials(Request))):
    if key==None:
        return {"message":"API key not validated"}
    else:
        return persons
```

If the `http://localhost:8000/persons` URL is visited without registering the **api_key** cookie, the browser shows the "API key not validated" message. However, if the cookies are set, the list of persons will be rendered (Listing 7-20).

Listing 7-20. Response after cookie validation

```
[
  {
    "name": "Tom",
    "age": 20
  },
  {
    "name": "Mark",
    "age": 25
  },
  {
    "name": "Pam",
    "age": 27
  }
]
```

Using Class As Dependency

Instead of a function (normal or async), it is possible to use a class. Let us turn the `properties()` function into a `properties` class and use it as a dependency (Listing 7-21).

Listing 7-21. Dependency class

```
class properties:
    def __init__(self, x:int, y:int):
        self.x=x
        self.y=y
```

To use it as a dependency, put the name of the class as an argument to the `Depends()` function (Listing 7-22).

Listing 7-22. Using a class for dependency injection

```
@app.get("/persons/")
async def get_persons(params: properties =
Depends(properties)):
    return persons[params.x:params.y]
```

Note that the `Depends()` function now returns an object of the `properties` class rather than a dict in the earlier example. The instance variables `x` and `y` are used to slice the `persons` list.

You can also have a callable class as the dependency. In Python, a callable class is the one that overrides the `__call__()` method, one of Python's magic methods. As a result, the object of such a class acts as a callable, that is, a function.

Let us change the `properties` class as in Listing 7-23.

Listing 7-23. Callable class as a dependency

```
class properties:
    def __call__(self, x:int, y:int):
        self.x=x
        self.y=y
        return self
```

Inside the operation function, the `Depends()` function in Listing 7-24 uses the object of this class as the argument.

Listing 7-24. Using a callable object as a dependency

```
@app.get("/persons/")
async def get_persons(params: properties =
Depends(properties())):
    return persons[params.x:params.y]
```

```
@app.get("/employees/")
async def get_employees(params: properties =
Depends(properties())):
    return employees[params.x:params.y]
```

Database Session Dependency

In the previous chapter (**Chapter 6 – Using Databases**), we had used the dependency injection to make the database context available to the operation functions.

The code of the GET operation in the “SQLAlchemy” section of the previous chapter is repeated here (Listing 7-25).

Listing 7-25. Generator function as a dependency

```
def get_db():
    db = session()
    try:
        yield db
    finally:
        db.close()

@app.get('/books', response_model=List[Book])
def get_books(db: Session = Depends(get_db)):
    recs = db.query(Books).all()
    return recs
```

To fetch the records from a database table, the `get_books()` function needs the database session object. It is injected by the `get_db()` function here.

The `get_db()` dependency function appears to be a normal Python function. However, instead of `return`, it uses a `yield` statement. Although `yield` is similar to `return` in the sense that it also returns a value to the

calling environment, there is a major difference between the two. The `yield` statement returns a generator object, but doesn't exit the function. This allows a certain cleaning operation, such as closing the connection after the path operation delivers its response.

A dependency function may also use `yield` inside Python's context manager syntax, as in Listing 7-26.

Listing 7-26. Dependency using a context manager

```
def get_db():
    with session() as db:
        yield db
```

Dependency in Decorator

Sometimes, the return value of the dependency function is not required to be injected into the operation function. But still, you need that dependency to be solved. In other words, the dependency functions must be executed before the path operation.

To do so, declare the dependency in the path decorator.

For instance, you want to ensure that the client request contains a specific custom header. The dependency function (Listing 7-27) raises an exception if the required header is not detected.

Listing 7-27. Path operation to insert Header

```
async def verify_header(X-Web-Framework: str = Header()):
    if X-Web-Framework != "FastAPI":
        raise HTTPException(status_code=400, detail="Invalid
        Header")
```

Note that this function doesn't return any value. Hence, it doesn't inject any parameter in the operation function. However, we can force this dependency to be solved by defining the `Depends()` parameter in the decorator itself, as in Listing 7-28.

Listing 7-28. Decorator with a dependency

```
from fastapi import Depends, FastAPI, Header, HTTPException

app = FastAPI()

@app.get('/books', dependencies=[Depends(verify_header)])
def get_books(db: Session = Depends(get_db)):
    recs = db.query(Books).all()
    return recs
```

If there are more than one dependency, they can be put in a list. While this applies to a given path operation, you can very well enforce a global dependency across all the routes in an application. To do so, use `Depends()` as an argument in the FastAPI object constructor (Listing 7-29).

Listing 7-29. Global dependency

```
app = FastAPI(dependencies=[Depends(verify_header)])

@app.get('/books')
def get_books(db: Session = Depends(get_db)):
    recs = db.query(Books).all()
    return recs
```

In a subsequent chapter, we shall see how to apply dependencies to enforce security requirements and authentication in a FastAPI app.

Middleware

A middleware is a function that intercepts every HTTP request before being processed by the corresponding path operation function. If required, the function can use the request to perform some process. The request object is then handed over to the path operation function. The middleware can also modify the response before it is rendered.

The middleware function is decorated by `@app.middleware("http")`. It receives two arguments. The first one is the client request, and the second is a `call_next` function.

The `call_next` function passes the request to the intended path operation function. Its response may be manipulated by the middleware before returning to the client.

The simple example (Listing 7-30) shows how the middleware works. The code has a single route `"/` that displays a Hello World message. The `add_header()` function intercepts the request, prints a text message on the console, and passes the request to the path operation function. Before rendering the response, it adds an **X-Framework** response header.

Listing 7-30. Custom middleware

```
from fastapi import FastAPI, Request, Header
from typing import Optional
app = FastAPI()

@app.middleware("http")
async def add_header(request: Request, call_next):
    print ("Message by Middleware before operation function")
    response = await call_next(request)
    response.headers["X-Framework"] = "FastAPI"
    return response
```

```
@app.get("/")
async def index(X_Framework: Optional[str] = Header(None)):
    return {"message": "Hello World"}
```

If you check the documentation of the app, the server's response reveals the header inserted by the middleware (Figure 7-9).

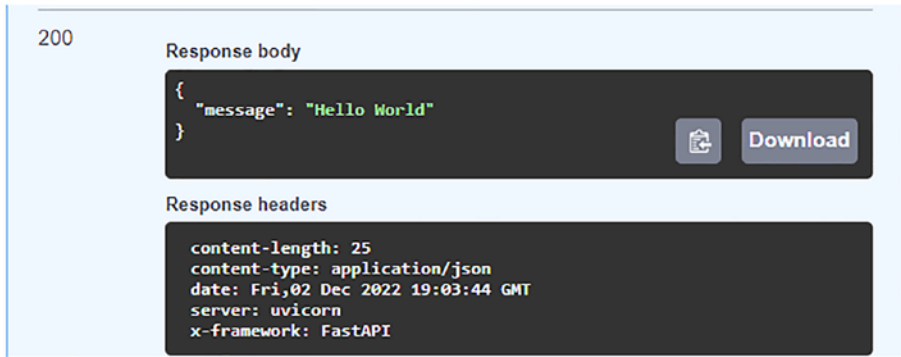


Figure 7-9. Header inserted by middleware

The following middleware are integrated into the FastAPI library:

HTTPSRedirectMiddleware: Enforces that all incoming requests must either be https or wss. Any incoming requests to http or ws will be redirected to the secure scheme instead.

TrustedHostMiddleware: Ensures that all incoming requests have a correctly set Host header to prevent HTTP Host Header attacks.

GZipMiddleware: Handles GZip responses for any request that includes "gzip" in the Accept-Encoding header.

CORS

The term CORS stands for **Cross-Origin Resource Sharing**. Imagine a situation where a certain front-end application running on www.xyz.com is trying to communicate with a back-end application running on www.abc.com. Here, the front-end and back-end applications are on different “origins.” Browsers normally restrict such cross-origin requests.

FastAPI’s **CORSMiddleware** makes it possible to accept URL requests from certain domains whitelisted in the application.

To configure the FastAPI app for CORS, import **CORSMiddleware** and specify the allowed origins (Listing 7-31).

Listing 7-31. Using CORSMiddleware

```
from fastapi.middleware.cors import CORSMiddleware

origins = [
    "http://localhost"
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

By default, the Uvicorn server accepts the incoming requests at port 8000 of the localhost. The allowed list of origins is localhost only (not specifying the port means it is 80 by default).

The FastAPI app in Listing 7-32 has just a “/” route.

Listing 7-32. FastAPI app

```

from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def main():
    return {"message": "Hello World"}

```

Of course, the application runs fine on the Uvicorn server at localhost:8000. But we want to send a request to its `/` URL from the localhost:80 domain.

To do so, create a simple HTML script (**hello.html** as in Listing 7-33) in the web root location of the **Apache** server.

Listing 7-33. Front end of the allowed origin

```

<html>
<body>
<a href="http://localhost:8000/">Click here</a>
</body>
</html>

```

While your FastAPI app is running, launch the Apache server and go to `http://localhost:80/hello.html`. Click the hyperlink. The browser is redirected to `http://localhost:8000/` which is the URL of your FastAPI app. You will see the Hello World message displayed.

Summary

Some very important features of FastAPI have been explained in this chapter. We learned how to build large-scale applications with APIRouter. We also dived deep into the concept of dependencies and middleware.

In the next chapter, some more advanced features of FastAPI are going to be discussed. This includes **WebSockets**, **GraphQL**, and others.

CHAPTER 8

Advanced Features

You now know enough about how to build REST APIs with FastAPI. In this chapter, you will learn to use modern web applications with WebSocket and GraphQL technology. We shall also explore the events in FastAPI and how to include a Flask application.

This chapter is arranged in the following topics:

- WebSockets
- WebSockets module in FastAPI
- Test WebSockets with Insomnia
- Multiclient chat application
- GraphQL
- FastAPI events
- Mounting WSGI application

WebSockets

The HTTP protocol is the backbone of the Internet. We know that HTTP is a stateless protocol. After sending its response, the server doesn't hold back any details about the client. Hence, the client has to reestablish the connection with the server for each subsequent interaction.

The WebSocket protocol, even though it works on top of HTTP, provides a two-way communication channel (full duplex) over a single TCP connection. Since the connection doesn't get disconnected after every transaction, the WebSocket protocol is ideally suited for real-time applications.

The WebSocket protocol has introduced **ws://** and **wss://** as the new URI schemes. In other words, instead of **http://** (or **https://** in the case of secured HTTP), the WebSocket URL is prefixed by **ws://** (or **wss://** for WebSocket Secure).

How Do WebSockets Work?

As mentioned earlier, the WebSocket protocol establishes a persistent, bidirectional TCP connection between the server and the client. As a result, a real-time communication between the two becomes possible (Figure 8-1).

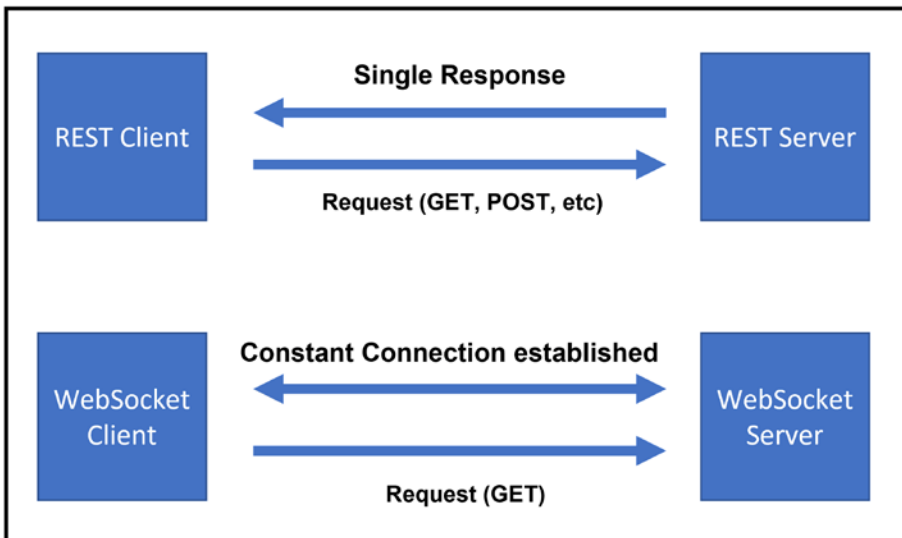


Figure 8-1. *REST vs. WebSocket*

To establish a WebSocket connection, the client browser sends a WebSocket handshake request to a server to upgrade the connection, along with a **Sec-WebSocket-Key** header. The server acknowledges the handshake and inserts a hash of the key in a **Sec-WebSocket-Auth** header.

Once the connection is established, it doesn't need to follow the HTTP protocol. A server application communicates with each client individually. Since WebSocket is a persistent connection, the server and the client can continue to exchange messages for any length of time until one of them chooses to close the session. In the WebSocket communication, event-driven web programming is possible. The standard HTTP allows only the clients to request new data. When using WebSocket though, two-way communication is established.

A WebSocket server can be written in any server-side programming language such as Python. To build a WebSocket server and client using Python, you need to install the `websockets` library. Since it is built on top of Python's `asyncio` package, you should use Python's 3.4 version or later.

You can install the `websockets` library with PIP. If you have installed **uvicorn** with a standard option, the `websockets` library has already been present in your working Python environment:

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

WebSocket Server

To start up a WebSocket server, call the `serve()` coroutine defined in the `websockets` module and provide **ws_handler**, host, and port as arguments, for example:

```
server = websockets.serve(hello, 'localhost', 8765)
```

Here, `hello()` is the WebSocket handler coroutine.

Run the server till the WebSocket handler coroutine is completed, with the statements shown in Listing 8-1.

Listing 8-1. asyncio loop

```
ws=asyncio.get_event_loop()
ws.run_until_complete(server)
ws.run_forever()
```

Now, the handler coroutine is given in Listing 8-2. It basically waits for the client to request connection, consumes the data it receives, and then sends back its response.

Listing 8-2. Server-side code

```
import asyncio
import websockets

async def hello(websocket, path):
    name = await websocket.recv()
    print("< {}".format(name))

    greeting = "Hello {}!".format(name)
    await websocket.send(greeting)
    print("> {}".format(greeting))
```

WebSocket Client

On the client side too, you need a coroutine that sends a connection request to the WebSocket server and asynchronously sends certain data. As the server responds after acknowledging the request, the client processes the incoming data. Here's the client-side coroutine (Listing 8-3).

Listing 8-3. Client code

```
import asyncio
import websockets

async def hello():
```

async with websockets.connect('ws://localhost:8765') as
websocket:

```

    name = input("What's your name? ")
    await websocket.send(name)
    print("> {}".format(name))

    greeting = await websocket.recv()
    print("< {}".format(greeting))
loop=asyncio.get_event_loop()
loop.run_until_complete(hello())

```

All you have to do on the client side is to run an asyncio loop till the coroutine is complete.

Run the server and client code in two separate command terminals. Input a string in the client window. There should be an immediate response from the client:

python ws-client.py

```

What's your name? test
> test
< Hello test!

```

WebSockets Module in FastAPI

FastAPI has an integrated support of the **WebSocket** protocol. This makes it easy to develop real-time web applications that stream data to be consumed by web, desktop, or mobile device clients.

Just as the HTTP path operations are handled by the path operation decorators (such as `@app.get()`, `@app.post()`, etc.), the FastAPI app object defines the `@app.websocket()` decorator. It starts a WebSocket server that listens for incoming requests at the URL path given as a string parameter.

The coroutine defined below (in Listing 8-4) this decorator handles the WebSocket protocol.

Listing 8-4. WebSocket decorator

```
from fastapi import FastAPI, WebSocket

app = FastAPI()
@app.websocket("/test")
```

The `fastapi.WebSocket` module provides the necessary functionality to send and receive data to and from the client. The WebSocket path operation function, invoked whenever a client visits the URL with the **ws://** or **wss://** protocol, accepts the incoming request and processes the incoming data with the `async receive()` method (there are other variations of the `receive()` method, such as `receive_text()` and `receive_json()` methods). It may send the data back to the client with `async send()` – there are `send_text()` and `send_json()` methods as well.

Let us define a simple `async` function to handle the WebSocket connection request as in Listing 8-5.

Listing 8-5. URL route for WebSocket

```
@app.websocket("/test")
async def test(websocket: WebSocket):
    await websocket.accept()
    while True:
        request = await websocket.receive_text()
        print(request)
        while True:
            i=random.randint(1,1000)
            await websocket.send_text(str(i))
            if i==100:
                break
```

After acknowledging the request, this function runs a loop and sends a series of random numbers.

What happens on the client side? The client app may be a **React** or an **Angular** application or a mobile app that communicates with the back end in the native code.

For the sake of simplicity, we'll use a simple web page as the client. We also have some JavaScript code in it to handle the WebSocket communication.

To open a new WebSocket connection within a JavaScript code, use the special protocol **ws** in the URL:

```
var ws = new WebSocket("ws://localhost:8000/test")
```

Remember, our FastAPI app will run the WebSocket server at the / **test** URL.

Once the socket is established, we need to listen to the following events on it:

- **open**: Connection established on accepting the request
- **message**: Data received by either the server or the client
- **error**: WebSocket error
- **close**: When the connection is closed by either the client or the server

There is a button in the HTML page. Its **onclick** event calls the `handleOnClick()` function to send a connection request to the WebSocket set up by our FastAPI code. With every message sent by the server, the client receives the random numbers, which are rendered on the page. Save the HTML script (Listing 8-6) as `test.html`.

Listing 8-6. WebSocket client in JavaScript

```

<script>
  var ws = new WebSocket("ws://localhost:8000/test")
  ws.onmessage = event => {
    var number = document.getElementById("number")
    number.innerHTML = event.data
  }
  handleClick = () => {
    ws.send("Hi WebSocket Server")
  }
</script>
<h3>Streaming numbers appear here</h3>

<div id="number"></div>
<button onclick="handleOnClick()">Click Me</button>

```

Run the Uvicorn server, and then open the test.html in a browser. As you click the button in the browser, the test() function in FastAPI code fires and starts sending random numbers. The numbers start appearing on the web page until the number generated happens to be 100.

Let us look at a little more elaborate example in Listing 8-7.

The “/” route on the FastAPI server-side code reads a **socket.html** file and renders a form.

Listing 8-7. Rendering an HTML form

```

@app.get("/", response_class=HTMLResponse)async def
hello(request: Request):
    file=open("templates/socket.html")
    html=file.read()

```

On the HTML form (Listing 8-8), we have a textbox for sending a text input to the WebSocket client.

Listing 8-8. Client-side form

```

<h1>WebSocket Client</h1>
<form action="" onsubmit="sendMessage(event)">
  <input type="text" id="sendText"/>
  <input type="submit" name="send">
  <button onclick="handleOnClick()">Close</button>
</form>
<ul id='messages'>
</ul>

```

Note that **** element with `messages` as its Element Id is used to echo the sent messages.

The event handler the JavaScript function attached to the Send button is given in Listing 8-9.

Listing 8-9. JavaScript function to send messages

```

function sendMessage(event) {
  var input = document.getElementById("sendText")
  ws.send(input.value)
  input.value = ''
  event.preventDefault()
}

```

The server just echoes back the received text. Listing 8-10 has the code for the WebSocket handler coroutine.

Listing 8-10. WebSocket handler function

```

from fastapi import WebSocket

@app.websocket("/ws")
async def ws_handler(websocket: WebSocket):
  await websocket.accept()

```

```

while True:
    data = await websocket.receive_text()
    await websocket.send_text(f"Message text was: {data}")

```

The JavaScript code on the client (Listing 8-11) is listening to the incoming messages. They are appended to the `` element.

Listing 8-11. Message listener in the JavaScript client

```

ws.onmessage = function(event) {
    var messages = document.getElementById('messages')
    var message = document.createElement('li')
    var content = document.createTextNode(event.data)
    message.appendChild(content)
    messages.appendChild(message)
};

```

Finally, the Close button is provided to disconnect the WebSocket connection (Listing 8-12).

Listing 8-12. Close WebSocket connection

```

handleOnClick = () => {
    ws.close();
    alert("Connection Closed");
}

```

While the FastAPI app is running, visit `http://localhost:8000/` to open a chat window. Figure 8-2 shows the alert box if the Close button is clicked.

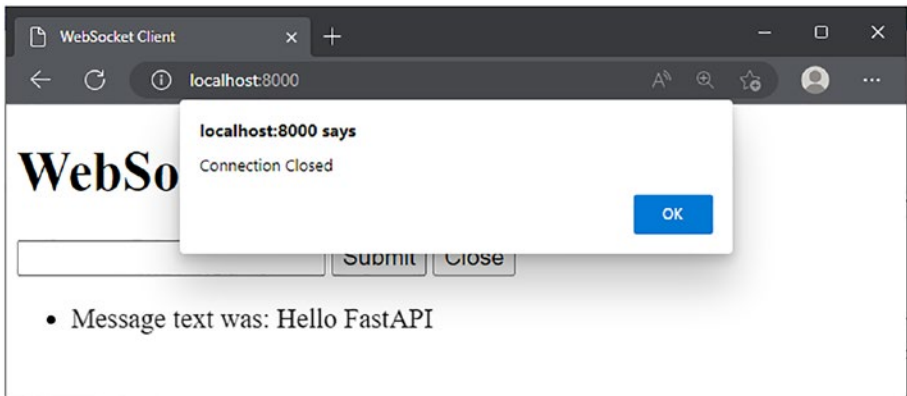


Figure 8-2. *WebSocket client in action*

Test WebSockets with Insomnia

In addition to the use of a front-end application, such as the one used in the preceding example, there are many utilities with which the WebSocket server communication can be tested. Developers often use apps such as Postman and Insomnia to test the WebSocket API endpoints.

Insomnia is a cross-platform desktop application to test REST, WebSocket, and GraphQL APIs. It has an easy-to-use interface, and it's free for anyone to use. It is available for download at <https://insomnia.rest/download>.

To test the `/ws` endpoint defined in the previous example, create a new WebSocket request from the Insomnia GUI, enter the URL `http://localhost:8000/ws`, and click the Connect button. You can then send the messages and obtain the response. An example of the WebSocket communication session is shown in Figure 8-3.

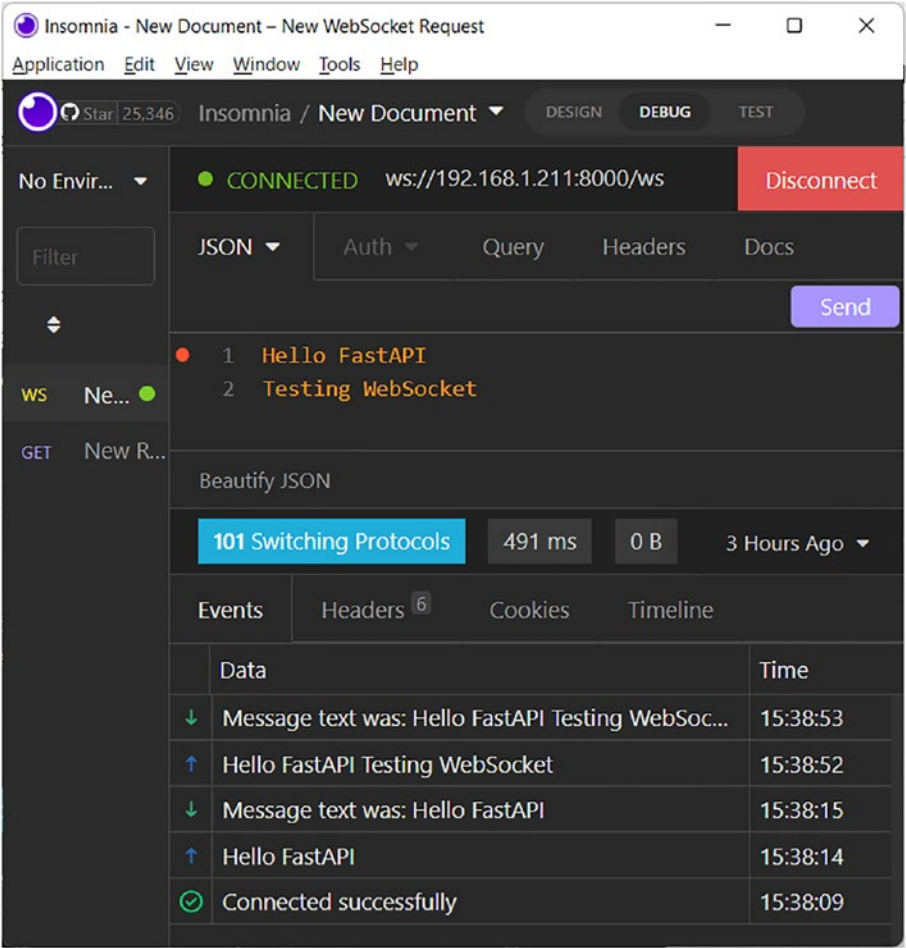


Figure 8-3. *Insomnia WebSocket request*

Multiclient Chat Application

The fact that WebSocket is a bidirectional communication protocol makes it ideal for building real-time applications. In the example for this section, we shall extend the preceding example and implement a chat with

multiple clients interacting simultaneously. The message submitted by one client is broadcast to all the open connections. We shall also handle the event of socket disconnection in this example.

The functionality of maintaining the list of currently live connections and sending and broadcasting the message is handled by the `Connection_handler` class in Listing 8-13.

Listing 8-13. `Connection_handler` class

```
class Connection_handler:

    def __init__(self):
        self.connection_list: List[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.connection_list.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.connection_list.remove(websocket)

    async def send_message(self, message: str, websocket:
        WebSocket):
        await websocket.send_text(message)

    async def broadcast(self, message: str):
        for connection in self.connection_list:
            await connection.send_text(message)
```

Further, the application code (Listing 8-14) uses `connection_list` to maintain a list of all the `WebSocket` client objects. Every time the connection request is accepted from a client object, it is appended to the list. Similarly, as the client disconnects, it is removed from the list. The `broadcast()` method sends the message to all the currently available clients in the list.

The `@app.websocket()` decorator in the code parses the path parameter from the request URL into the `client_id` and passes it to the operation function beneath it.

Inside the function, the incoming connection request is accepted, and the message is sent to all the active connections. The `Connection_handler` also updates its list of active connections.

As and when the server detects that a client has disconnected, all the other clients get notified accordingly. Listing 8-14 shows the code for the WebSocket operation function.

Listing 8-14. WebSocket handler coroutine

```
manager = Connection_handler()

@app.websocket("/ws/{client_id}")
async def websocket_endpoint(websocket: WebSocket, client_id: int):
    await manager.connect(websocket)
    try:
        while True:
            data = await websocket.receive_text()
            await manager.send_message(f"You wrote: {data}",
                                      websocket)
            await manager.broadcast(f"Client #{client_id} says: {data}")
    except WebSocketDisconnect:
        manager.disconnect(websocket)
        await manager.broadcast(f"Client #{client_id} left the chat")
```

The `hello()` function in the FastAPI code for the `"/"` URL route is kept as it was in the earlier example.

We need to modify the JavaScript code in our `socket.html` (Listing 8-15). We need to generate a random client ID, display it on the chat page, and include it as the parameter in the request URL.

Listing 8-15. Obtaining a client ID

```
var client_id = Math.floor(Math.random() * 100);
document.querySelector("#wsID").textContent = client_id;
var ws = new WebSocket(`ws://localhost:8000/ws/${client_id}`);
```

The rest of the JavaScript functionality remains the same. In the HTML body section, provide a span element with **wsID** as its ID, as in Listing 8-16.

Listing 8-16. Modified client form

```
<h1>WebSocket Client</h1>
  <h2>Your ID: <span id="wsID"></span></h2>
  <form action="" onsubmit="sendMessage(event)">
    <input type="text" id="sendText"/>
    <input type="submit" name="send">
    <button onclick="handleOnClick()">Close</button>
  </form>
  <ul id='messages'>
  </ul>
```

Run Uvicorn to serve the FastAPI application. Open multiple browser windows and enter the `http://localhost:8000/` URL in each. Each window shows a unique, random client ID (Figure 8-4). Try sending messages from each window and observe them broadcast to all. From one of the browsers, click the **Close** button (or close the window itself). The remaining open windows will get the notification of a client having left.

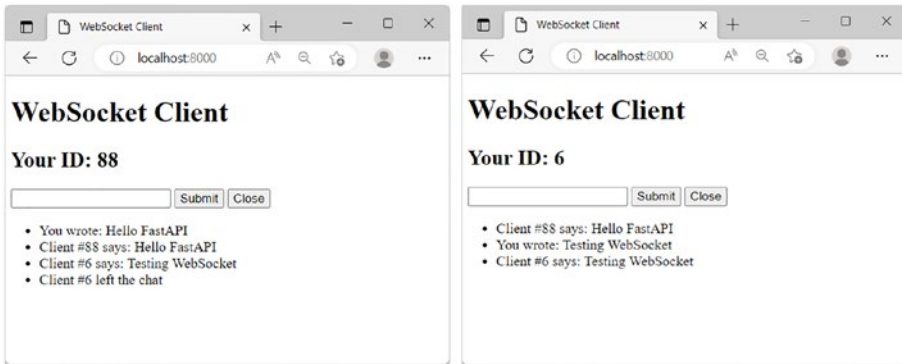


Figure 8-4. Multiclient chat with WebSockets

GraphQL

Even though REST is widely regarded as the standard for designing web APIs, the REST APIs tend to be too rigid and hence can't handle the clients' requirements that keep changing.

GraphQL was developed to address the shortcomings and inefficiencies of the REST APIs and provide more flexible and efficient alternative.

A major drawback of REST is that a client downloads more information than is actually required for its consumption. The response from an API endpoint might contain much more info that is superfluous for the client, as it might need only some details of the resource.

Another issue with REST is that a specific endpoint doesn't provide enough information required by the client. Hence, the client might need to make additional requests to fulfill the client's requirements.

GraphQL has been developed and open-sourced by **Facebook**. Because of its efficiency and flexibility, GraphQL is now recognized as a new API alternative to REST architecture.

With its declarative data fetching, the GraphQL client is able to ask for exactly what data it needs from an API. Another feature of GraphQL is that a GraphQL server only exposes a single endpoint as against REST where multiple endpoints are defined. Hence, instead of returning fixed data structures, the GraphQL server responds with the data precisely as per client request.

Contrary to a common misconception, GraphQL is not a database technology. GraphQL is a query language for APIs and has nothing to do with SQL which is used by databases. In that sense, it is independent of any type of database. You can use it in any context where you need an API.

Figure 8-5 illustrates the GraphQL architecture.

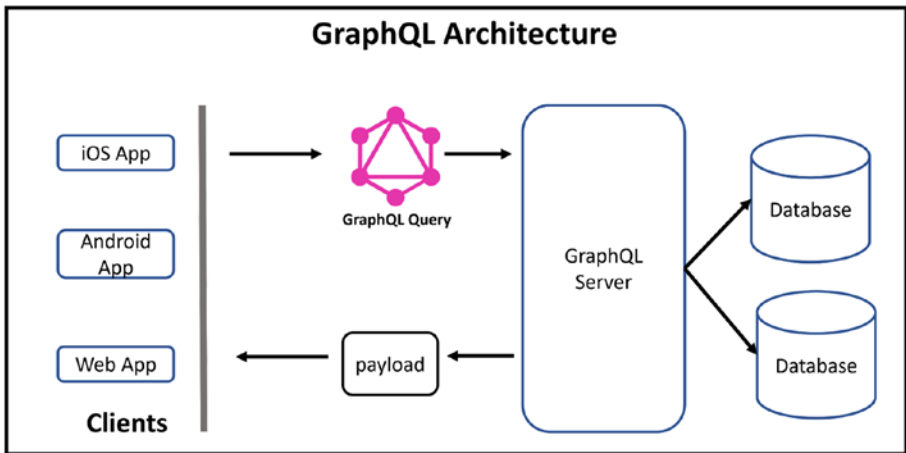


Figure 8-5. *GraphQL architecture*

The Schema Definition Language

As a first step, you need to define the schema of an API you are going to develop. GraphQL uses the Schema Definition Language (**SDL**) to write the API schema.

The example in Listing 8-17 shows how to use the SDL to define the Book type.

Listing 8-17. Type definition in GraphQL

```
type Book {  
  title: String!  
  price: Int!  
}
```

We have defined two fields in the `Book` type here; they're called `title` (which is a string field) and `price` (which is an int field). In SDL, the `!` symbol after the type indicates that this field is required and cannot be null.

As mentioned earlier, a GraphQL API exposes only a single endpoint. On the contrary, a REST API has multiple endpoints that return fixed data structures. This works because the data structure returned is not fixed. In fact, it is totally flexible in nature. Therefore, the client can determine exactly what its requirements are.

However, to define its requirements precisely, the client has to provide more information to the server in the form of a query.

Queries

Here is an example of the query that can be sent by a client to a server (Listing 8-18).

Listing 8-18. GraphQL query

```
{  
  books {  
    title  
  }  
}
```

The `books` field in this query is called the root field of the query. The `title` field in the root field is called the **payload** of the query. This query would return a list of all books currently available.

Here, the response contains only the title of each book, but the price is not included in the response returned by the server. That's because GraphQL allows you to specify exactly what is required. Here, `title` was the only field that was included in the query.

You only have to include another field in the query's payload whenever needed (Listing 8-19).

Listing 8-19. Query with payload

```
{
  books {
    title
    price
  }
}
```

Mutations

Most applications invariably need to modify the data that's currently available with the back end. GraphQL uses mutations to make such changes. There are three types of mutations:

- To create or add new data
- To update or modify data
- To delete data

Mutations appear similar in structure as queries, but you must use the **mutation** keyword in the definition. To create a new Book, the mutation can be as shown in Listing 8-20.

Listing 8-20. GraphQL mutation

```
mutation {  
  createBook(title: "Python Programming", price: 750) {  
    title  
    price  
  }  
}
```

Notice that the mutation type also has a root field – named `createBook`. It takes two arguments that specify the title and price of the new book.

Here is the server response for the preceding mutation:

```
"createBook": {  
  "title": "Programming in Python",  
  "price": 750,  
}
```

Subscriptions

In GraphQL terminology, there is the concept of subscriptions, so that a real-time connection to the server is available, and the client immediately gets information about important events.

If a client subscribes to an event, the server pushes the corresponding data to it, whenever that particular event then actually happens. Subscriptions don't follow a typical "request-response-cycle" (as is done by queries and mutations). They are a stream of data sent over to the client.

The subscription on events happening on the `Book` type is shown in Listing 8-21.

Listing 8-21. GraphQL subscription

```
subscription {  
  newBook {  
    title  
    price  
  }  
}
```

Schema

In GraphQL, the concept of schema is of much importance. How the client requests the data depends on the definition of schema. It is sort of a contract that both the server and the client abide by.

A GraphQL schema is a collection of GraphQL root types. The special root types (those described earlier) are included in the definition of the schema of an API:

```
type Query { ... }  
type Mutation { ... }  
type Subscription { ... }
```

The client requests use Query, Mutation, and Subscription types as the entry points for communication with the server.

Strawberry GraphQL

Many programming languages, including Python, support the implementation of the GraphQL communication pattern. The declarative nature of GraphQL's Schema Definition Language suits Python nicely. With newer Python versions providing the capability of asynchronous processing in the form of `asyncio`, Python is now a natural choice for many GraphQL developers.

There are quite a few Python libraries available for implementing the GraphQL protocol. Here are some of them:

- Ariadne
- Strawberry
- Tartiflette
- Graphene

The latest releases of FastAPI recommend the **Strawberry** library for working with GraphQL. Strawberry leverages Python’s dataclasses, type hints, and ASGI standards, all of which also form the foundations for FastAPI.

To start with, install the Strawberry package with the PIP command:

```
pip install strawberry-graphql[fastapi]
```

We now declare a Book class (Listing 8-22) with title, author, and price fields and use the `@strawberry.type` decorator on top of it.

Listing 8-22. Book type in Strawberry-GraphQL

```
import strawberry

@strawberry.type
class Book:

    title: str
    author: str
    price: int
```

Next, we need to add a Query type in our schema (Listing 8-23).

Listing 8-23. Query Book type in Strawberry-GraphQL

```
@strawberry.type
class Query:

    @strawberry.field
    def book(self) -> Book:
        return Book(title="The Godfather", author="Mario Puzo",
                    price=750)
```

The `book()` method in the `Query` class returns an instance of the `Book` class.

As mentioned earlier, the GraphQL schema consists of `Query`, `Mutation`, and `Subscription`. As of now, we have the `Query` class. Use it as an argument in the `Schema` constructor, and obtain the GraphQL application object from the schema (Listing 8-24).

Listing 8-24. GraphQL object

```
from strawberry.asgi import GraphQL

schema = strawberry.Schema(query=Query)
graphql_app = GraphQL(schema)
```

Finally, declare the `FastAPI` object, and add the URL route `"/book"`, (as in Listing 8-25) that invokes the callable `GraphQL` object.

Listing 8-25. FastAPI route for GraphQL

```
from fastapi import FastAPI

app = FastAPI()

app.add_route("/book", graphql_app)
```

Launch the FastAPI application on the Uvicorn server and visit the `http://localhost:8000/book` URL in your favorite browser. An in-browser tool called **GraphiQL** starts up (Figure 8-6). GraphiQL is a browser-based user interface with which you can interactively execute queries against a GraphQL API.

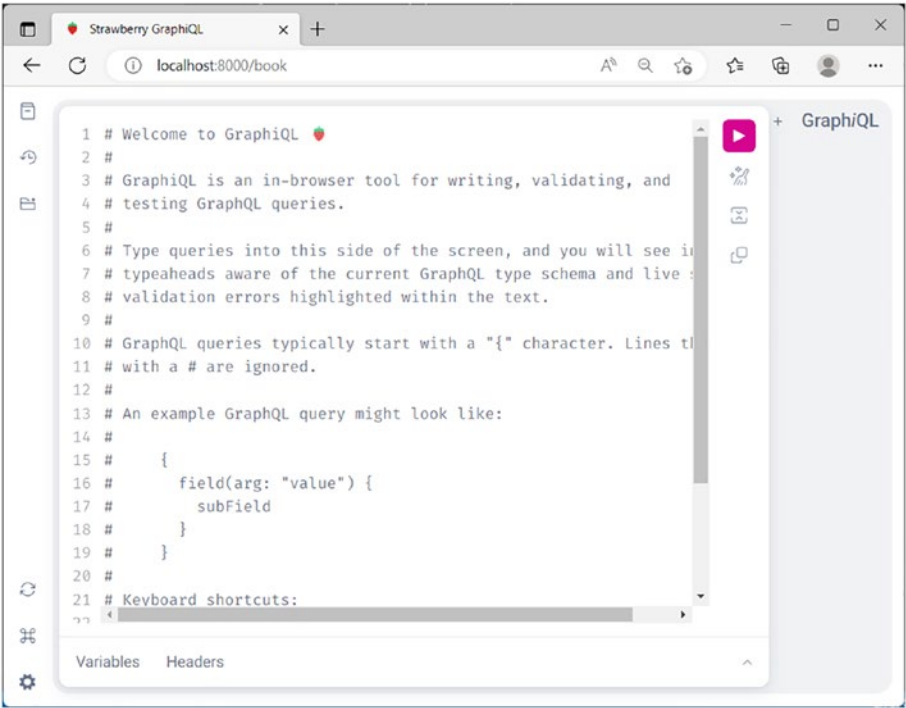


Figure 8-6. *GraphiQL IDE*

Scroll down below the commented section and enter the query in Listing 8-26 using the Explorer bar of the GraphiQL IDE.

Listing 8-26. Query in the GraphiQL IDE

```
query MyQuery {
  book {
    title
  }
}
```



```

    author
    price
  }
}

```

Run the query to display the result in the output pane of the IDE as shown in Figure 8-7.

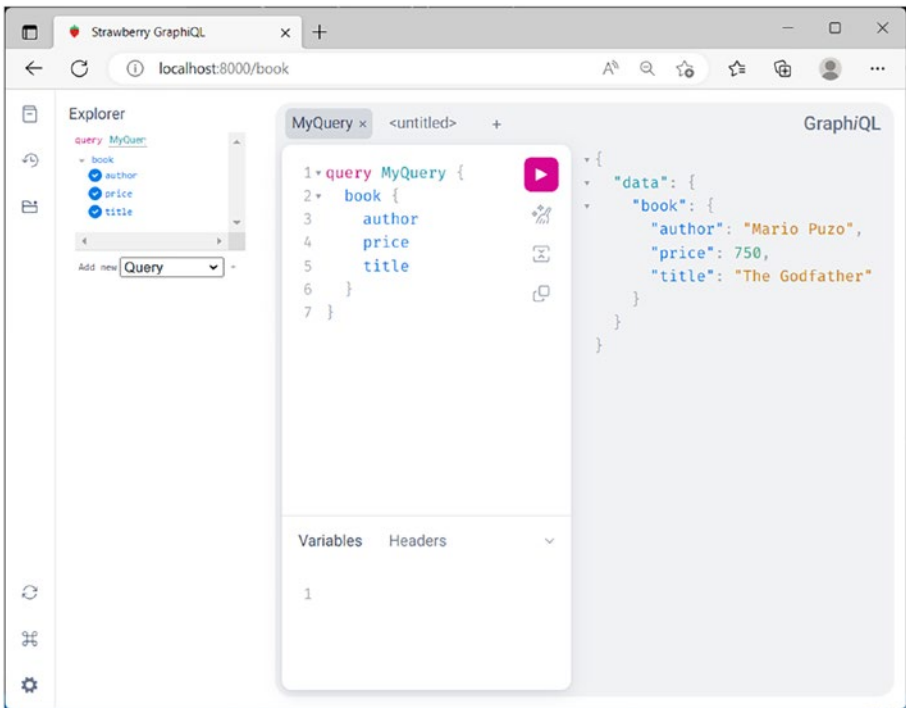


Figure 8-7. Query result in GraphQL

We have used earlier the Insomnia tool for testing WebSocket communication. Insomnia also supports testing GraphQL endpoints (Figure 8-8).

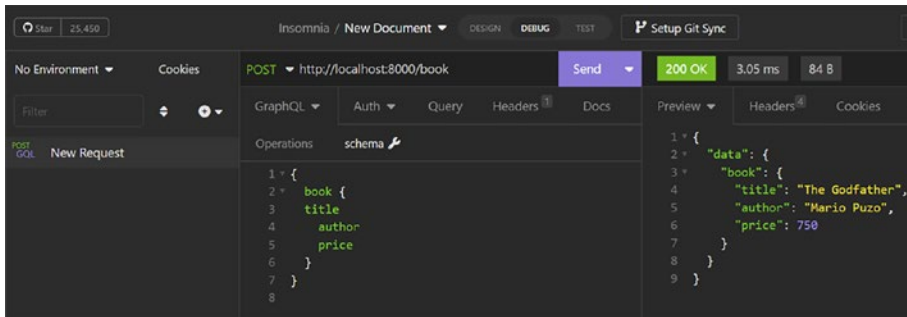


Figure 8-8. Insomnia for GraphQL testing

To perform create, update, and delete operations, we should add mutations to the GraphQL schema.

Add the Mutation class code in the FastAPI application, as in Listing 8-27.

Listing 8-27. Mutation class

```
@strawberry.type
class Mutation:
    @strawberry.mutation
    def add_book(self, title: str, author: str, price:int)
    -> Book:
        return Book(title=title, author=author, price=price)
```

When the browser is directed to the GraphQL endpoint **/book**, use the snippet in Listing 8-28 to run the mutation inside the **GraphiQL** IDE.

Listing 8-28. Parameters for the mutation function

```
mutation {
  addBook(
    title: "Harry Potter Collection",
```

```

    author: "J.K.Rowling",
    price:2500) {
  title
  author
  price
}
}

```

The output of GraphQL shows the result as in Figure 8-9.

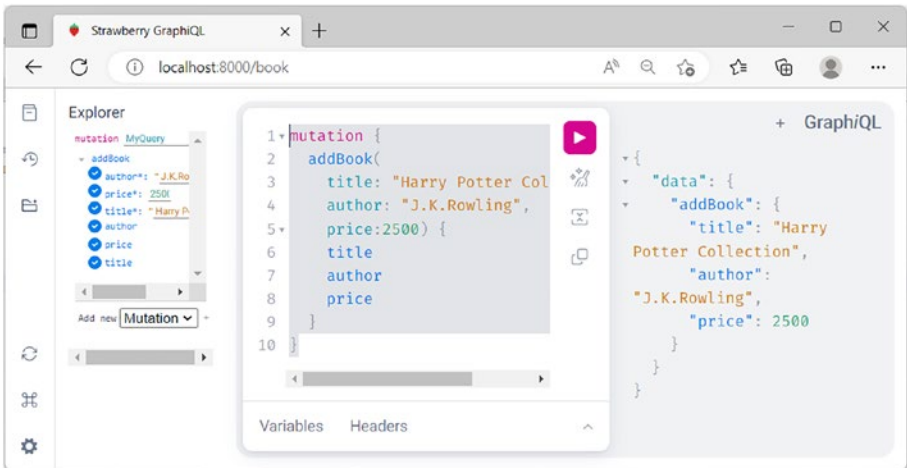


Figure 8-9. *GraphiQL of mutation*

You can always provide mutations for update and delete functionality. In typical use cases of GraphQL APIs, of course there will be a back-end database to perform these operations persistently. You can implement the database operations discussed in the earlier chapter of this book.

FastAPI Events

If you look carefully at the activities in the command terminal (in which you start the Uvicorn server), you will find that the event of starting the application as well as the shutting down is echoed there. You can attach a handler function to each of these two events. FastAPI has two decorators – `@app.on_event("startup")` and `@app.on_event("shutdown")` – for the purpose.

The decorated functions will fire as and when these events occur. Obviously, both events occur once during each run. This feature can be effectively utilized to perform a mandatory activity as the application starts and just before the server stops. A typical use case can be starting and closing the database connection.

In the example in Listing 8-29, a text file is being used by the FastAPI application to keep a log of the startup and shutdown time of the server. As the application starts, the startup time is recorded in the file. Similarly, when the user stops the server by pressing **Ctrl+C**, the shutdown time is appended to the file.

Listing 8-29 shows the application code.

Listing 8-29. FastAPI events

```
from fastapi import FastAPI
import datetime

app = FastAPI()

@app.on_event("startup")
def startup_event():
    print('Server started\n')
    log= open("log.txt", mode="a")
    log.write("Application startup at {}\n".format(datetime.
datetime.now()))
```

```

log.close()

@app.on_event("shutdown")
async def shutdown_event():
    print('server Shutdown :', datetime.datetime.now())
    log= open("log.txt", mode="a")
    log.write("Application shutdown at {}\n".format(datetime.
datetime.now()))
    log.close()

```

There will of course be other path operations defined in the code. However, every time the Uvicorn server starts, the `startup_event()` function writes the time in the file. Again, whenever the server is stopped, the timestamp is logged into the file:

```

Application startup at 2022-12-19 00:08:32.585333
Application shutdown at 2022-12-19 00:11:01.776408

```

Note that you should start the Uvicorn server in the production environment and not in debug mode (without the `--reload` flag); otherwise, the shutdown event will not be raised.

Mounting WSGI Application

Python's web application frameworks Flask, Django, etc., are widely used. You can use an existing Flask/Django application as a subapplication along with FastAPI. Flask and Django frameworks implement WSGI. To mount a Flask application, it should be wrapped with **WSGIMiddleware**.

Let us look at a basic example to understand how WSGIMiddleware works (Listing 8-30). Install flask and write a simple route that renders the Hello World message.

Listing 8-30. Flask subapplication

```

from flask import Flask
flask_app = Flask(__name__)

@flask_app.route("/")
def index_flask():
    return "Hello World from Flask!"

```

A similar path operation function for FastAPI is straightforward refer to Listing 8-31.

Listing 8-31. Main FastAPI application

```

from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def index():
    return {"message": "Hello World from FastAPI!"}

```

We now use the `WSGIMiddleware` class to wrap the Flask application object and mount it as the subapplication at the `/flask` endpoint (Listing 8-32).

Listing 8-32. Mounting the Flask app

```

from fastapi.middleware.wsgi import WSGIMiddleware

app.mount("/flask", WSGIMiddleware(flask_app))

```

The response from the main FastAPI application is served at the `http://localhost:8000/` URL:

```

{"message":"Hello World from FastAPI!"}

```

Visit <http://localhost:8000/flask> to obtain the message rendered by the Flask application:

```
Hello World from Flask!
```

Summary

With this, we are concluding the discussion of some of the advanced features of the FastAPI framework. We learned that FastAPI is not just about REST, and it supports the latest API technologies – namely, WebSockets and GraphQL. We now also know about the FastAPI events and how to use the Flask app with FastAPI.

In the next chapter, we shall explore the techniques to make our FastAPI code more robust and failproof, by learning about the security measures and how to run tests.

CHAPTER 9

Security and Testing

Since an API exposes the endpoints of the server's resources to be consumed by other third-party apps, ensuring that access is only granted to authorized users becomes the topmost priority for developers. Similarly, it is also important that the code is subjected to proper testing before releasing it to the users.

In this chapter, we are going to address these important topics:

- Exception handling
- Security
- Testing
- AsyncClient

Exception Handling

The fact that, in any software, a proper exception handling mechanism is of utmost importance cannot be overemphasized. An exception is a runtime error situation often leading to an abnormal termination of the process. Rather than leaving the user clueless about the reason of the termination, a proper feedback to inform the user about the reason of the exception is necessary.

In the case of an API, the exception may be caused by various reasons. For instance, the resource that the client (for an API, the client could be

a web browser, a desktop, a mobile application, etc.) is trying to access doesn't exist. Often, even if the resource is present, the client doesn't have necessary credentials to access it.

The HTTP protocol has an elaborate system of including an appropriate status code in the server's response. Since these situations (such as the resource doesn't exist) are client induced, the response includes a corresponding **4XX** status code.

Whenever the FastAPI operation function encounters a runtime error, it raises `HTTPException`. It inherits Python's `Exception` class, with an API-specific argument `status_code` that refers to the type of client error response.

Listing 9-1 shows a simple example of raising `HTTPException` in a FastAPI code.

Listing 9-1. `HTTPException`

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

names = [{"A01": "Alice"}, {"B01": "Bob"}, {"C01": "Christie"}]

@app.get("/names/{id}")
async def get_name(id: str):
    for name in names:
        if id in name.keys():
            return {"name": name[id]}
    else:
        raise HTTPException(status_code=404, detail="Name
        not found")
```

If the `get_name()` function finds that the `id` parameter obtained from the URL is indeed present in the `names` database (as in `http://localhost:8000/names/B01`), it renders a valid response to the client. If,

however, the id is not found, the `HTTPException` is raised. The string value of the detail parameter is used in the JSON response, as shown in Figure 9-1.

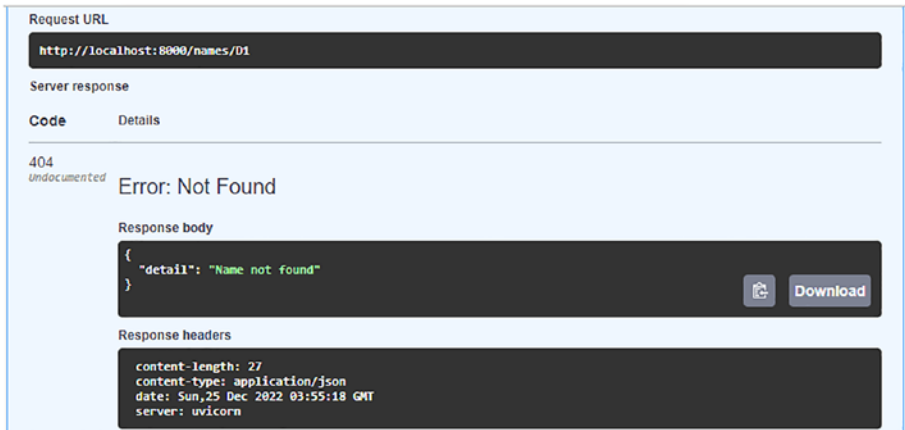


Figure 9-1. HTTP error response

User-Defined Exception

The Exception classes defined in FastAPI (in addition to the `HTTPException` discussed earlier, the `WebSocketException` class is also available) inherit Python’s Exception class. Hence, it is entirely possible to define a custom exception class, subclassing the Exception.

First, define a simple `MyException` class (Listing 9-2).

Listing 9-2. User-defined exception

```
class MyException(Exception):
    def __init__(self, msg:str):
        self.msg=msg
```

Now, we need to define an exception handler to process this type of exception occurring while the FastAPI object interacts with the client. This

exception handler function is decorated by `@app.exception_handler()`. It returns a JSON response with a status code of 406 (Not Acceptable) along with an appropriate error message (Listing 9-3).

Listing 9-3. Exception handler

```
@app.exception_handler(MyException)
async def myexceptionhanlder(request:Request, e:MyException):
    return JSONResponse(status_code=406, content={"message":
        "{} was encountered".format(e)})
```

Now we shall put this custom exception to use in our path operation function `get_name()` from the previous example. If the function doesn't find the name with the given id path parameter, we first check if it equals **'end'** and, if so, raise the newly defined `MyException`. If not, raise the `HTTPException` as done previously. Listing 9-4 provides the modified definition of `get_name()` function.

Listing 9-4. Raise a user-defined exception

```
@app.get("/names/{id}")
async def get_name(id: str):
    for name in names:
        if id in name.keys():
            return {"name": name[id]}
    else:
        if id=='end':
            raise MyException(id)
        else:
            raise HTTPException(status_code=404, detail="Name
                not found")
```

There are three possible scenarios here. First, the path operation is completed normally in response to the id parameter is present (Figure 9-2).



Figure 9-2. Normal path operation

Second, the id is not found, and it equals **end** (Figure 9-3).

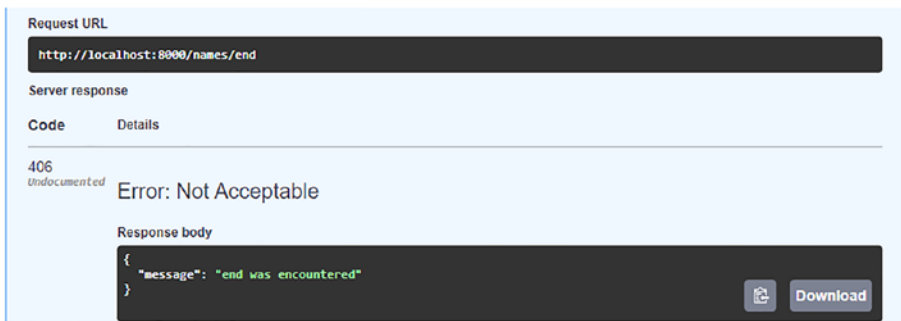


Figure 9-3. Error with status code 406

And third, the id is not found, but it's not end either, as in Figure 9-4.

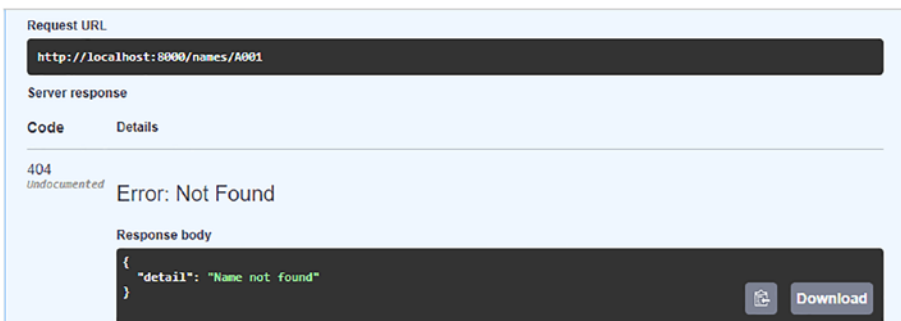


Figure 9-4. 404 error code

Security

We know that an API acts as the interface (API indeed is an acronym for **Application Programming Interface**) used by two different applications to transfer data. When an application grants access to the resources of its server to the outside world, the end users are authenticated and have the right kind of authorization. Securing an API is an important part of the development process, equally important if not more than the development of the application logic itself.

In this context, one often comes across two terms – authentication and authorization. Seemingly similar, they perform different functions and together provide a complete security system for an API. While authentication refers to verification of the identity of the user, authorization is the process that ascertains the permissions to the user.

Basic Access Authentication

A very basic authentication mechanism is provided by the HTTP protocol itself. First included in the HTTP 1.0 specification, it has since been superseded by **RFC 617** in 2015. The implementation of this authentication scheme requires the browser to send the username and password when it sends the request. **Base64** encoding is used to formulate the credentials. The request is packed with a header in the format **Authorization: Basic <credentials>**.

The HTTPBasic class is at the core of FastAPI's BA (Basic Access) authentication support. Its object is used as a dependency in the path operation function. An object of the class HTTPBasicCredentials contains the username and password provided by the client.

Have a look at Listing 9-5, which shows a simple path operation function with BA security dependency.

Listing 9-5. Basic security dependency

```
from fastapi import Depends, FastAPI
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()

scheme = HTTPBasic()

@app.get("/")
def index(logininfo: HTTPBasicCredentials = Depends(scheme)):
    return {"message": "Hello {}".format(logininfo.username)}
```

When a client browser goes to the “/” URL endpoint for the first time, a dialog box pops up as in Figure 9-5, prompting the user to provide the username and the password.

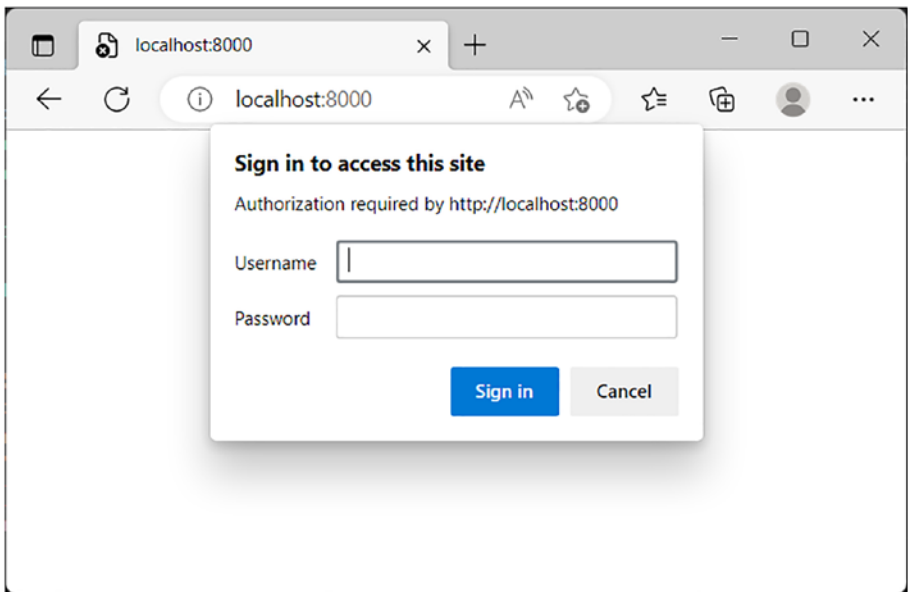


Figure 9-5. Basic authentication with a username and password

Although the password authentication logic has not been implemented here, you can always check it against a back-end database before returning the response.

If the authentication fails, the function may be made to return an `HTTPException` with the **401** status code.

OAuth

FastAPI has an out-of-the-box support for **OAuth2** security standard specification. OAuth stands for **Open Authorization**. OAuth version 2.0 provides simple authorization flows for web applications, desktop and mobile applications, etc.

One of the important features of OAuth is that it enables sharing information with another service without exposing your password. OAuth uses “access tokens.” An access token is a random string of alphanumeric characters. A bearer token (Figure 9-6) is the most commonly used. Once the OAuth client has the possession of the bearer token, it is able to make request for the associated resources with the server.

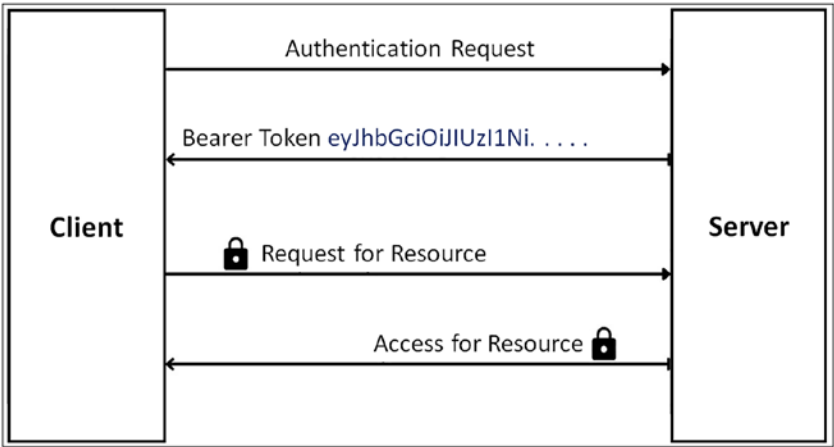


Figure 9-6. Bearer token

In the OAuth specification, the term “**grant type**” refers to the mechanism by which the application gets the access token. A grant type is also sometimes referred to as a flow. There are various grant types: authorization code, client credentials, implicit, and password. In this discussion, we shall discuss the password flow, which is the simplest way of implementing token authentication.

Let us start by defining a user **Pydantic** model which is mapped to a **SQLAlchemy** model named Users. In an earlier chapter, we have learned how the SQLAlchemy model structure is translated into a database table.

The definition of the Pydantic model and the corresponding SQLAlchemy model is shown in Listing 9-6.

Listing 9-6. User class – Pydantic and SQLAlchemy models

```
class User(BaseModel):
    id: int
    username: str
    password: SecretStr
    token: str

    class Config:
        orm_mode = True

class Users(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True, nullable=False)
    username = Column(String(50), unique=True)
    password = Column(String(200))
    token = Column(String(200))
```


It may be noted here that when using the password flow, the OAuth standard requires that the client must send the user data for validation as the values of **username** and **password** fields, without any other variation (which means you cannot have the model attributes as anything like `userId` or `user-name`, or the use of `pwd` instead of `password` is not accepted).

OAuth2PasswordBearer

Let us have a simple GET path operation that just renders a Hello World message (Listing 9-7). To enforce authentication, apply the `OAuth2PasswordBearer()` function as the dependency to the path operation function.

The `OAuth2PasswordBearer` class is defined in the `fastapi.security` module. Its constructor has a required argument in the form of `tokenUrl`, set to a URL route that returns the bearer token.

Listing 9-7. Enabling OAuth

```
from fastapi.security import OAuth2PasswordBearer

scheme = OAuth2PasswordBearer(tokenUrl='token')

@app.get('/hello')
async def index(token: str = Depends(scheme)):
    return {'message' : 'Hello World'}
```

With only this much of code in place, try to run the application. The Swagger UI page already shows its effect, with the **Authorize** button appearing prominently, as in Figure 9-7.

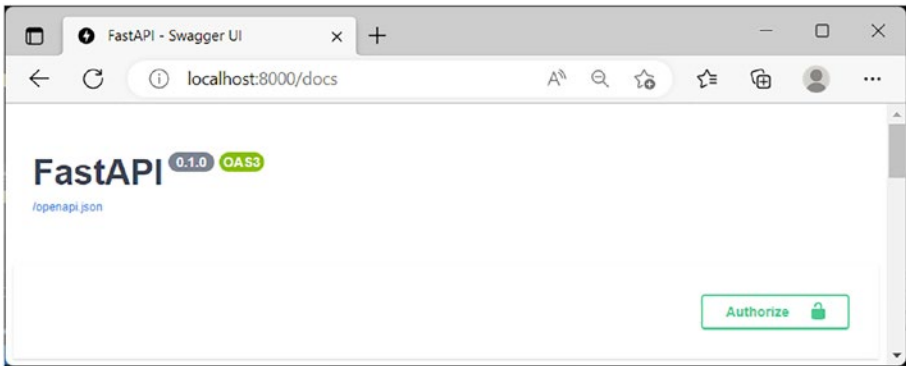


Figure 9-7. *OAuth enabled*

You also get to see the lock icon in front of the `index()` function. If you try to execute it, the server responds with a **401** status code, with a **Not Authenticated** message, as in Figure 9-8.

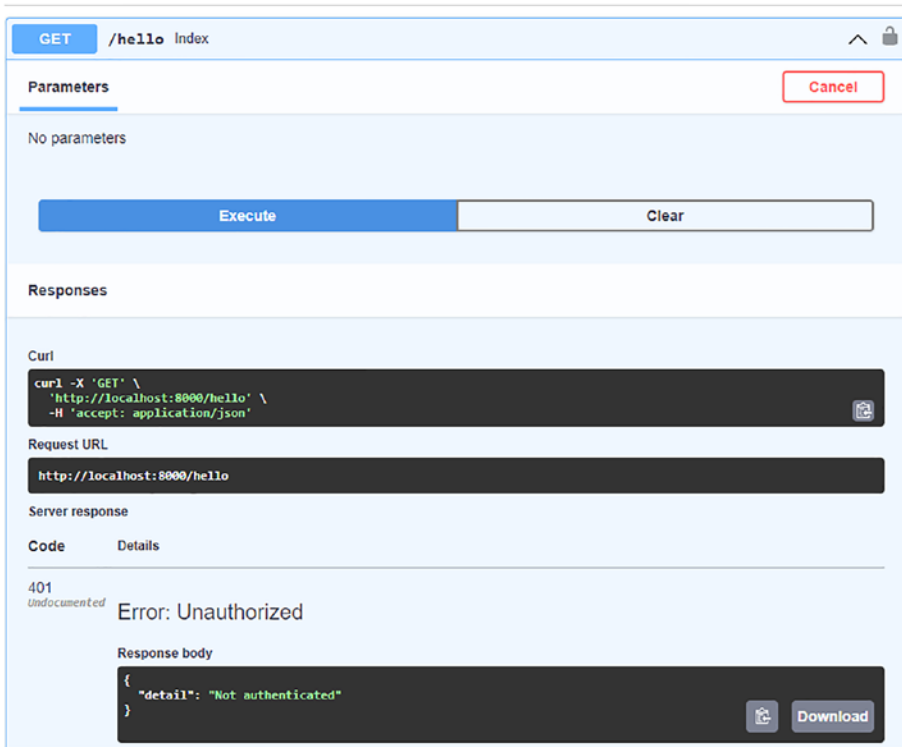


Figure 9-8. Server response with 401 code

The `index()` function (Listing 9-8) has a dependency as a callable object of the `OAuth2PasswordBearer` class. In turn, it invokes the `token()` function – a POST operation function associated with the `tokenUrl`.

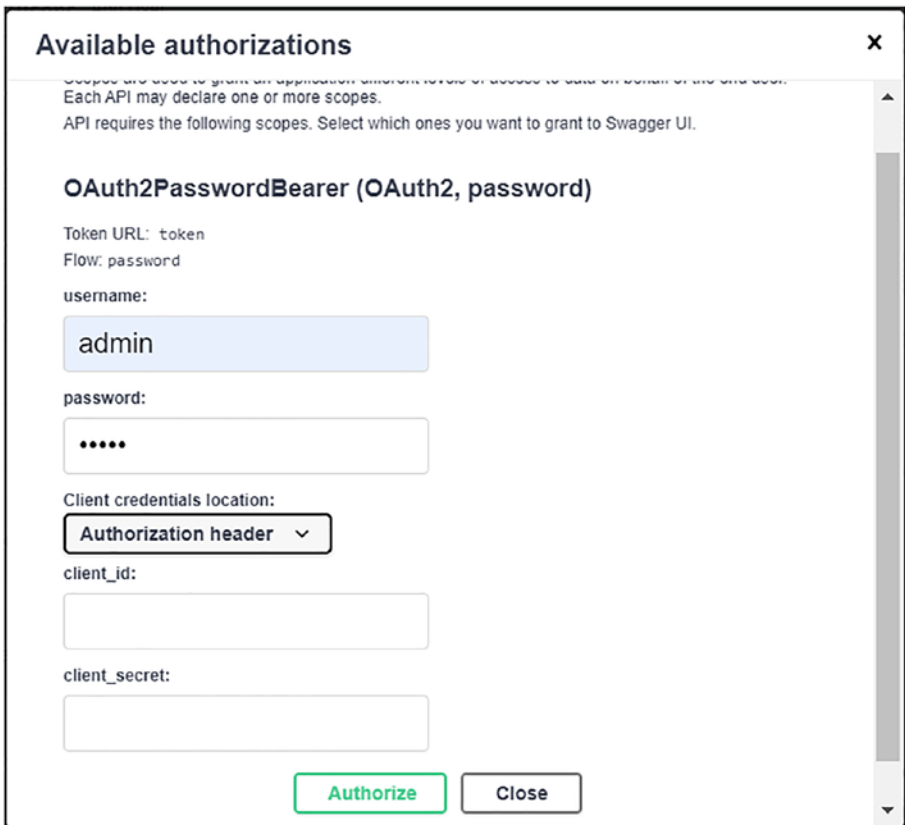
Listing 9-8. Function to invoke the password request form

```
@app.post('/token')
async def token(form_data: OAuth2PasswordRequestForm =
Depends()):
    token=hash(form_data.password)
    return {'Access Token' : token }
```

The preceding function also has `OAuth2PasswordRequestForm` as its dependency. It renders a form with username and password attributes. Taking these two body parameters, the `token()` function generates a token and returns it as the response.

You can write your own logic to generate a token. Here, we provide the password entered by the user to the `hash()` function. The hashed password is returned.

While on the Swagger UI page, click the lock icon. As shown in Figure 9-9, a form requesting the password pops up.



Available authorizations ✕

Scopes are used to grant an application permission to view or modify certain data on behalf of the end user. Each API may declare one or more scopes. API requires the following scopes. Select which ones you want to grant to Swagger UI.

OAuth2PasswordBearer (OAuth2, password)

Token URL: token
Flow: password

username:

password:

Client credentials location:
Authorization header ▼

client_id:

client_secret:

Authorize Close

Figure 9-9. *OAuth2PasswordRequestForm*

At this juncture, we are not applying any logic to authenticate the username and password. We shall check them against the database a little later. For now, the function just returns the hashed password when you click the **Authorize** button.

This also means that the user is now authorized to access the protected endpoints (Figure 9-10).

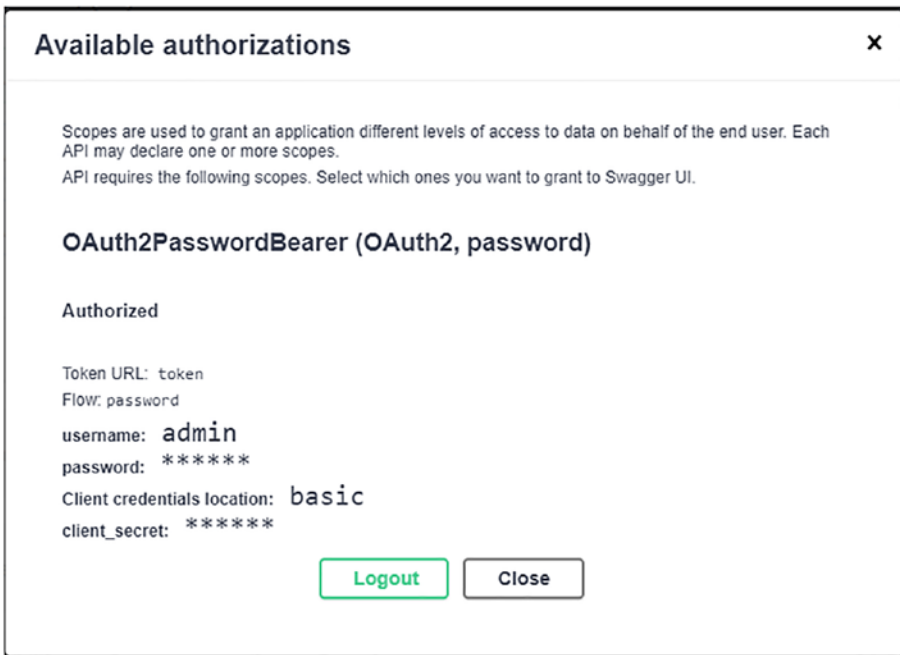


Figure 9-10. Access authorized

Go ahead and test the **/hello** route. Since the client now has the authorization token, you'll now see the Hello World message as its response.

Let us extend the example to provide the authentication of the username and password against the user data in a database (Listing 9-9). We have already defined the models. Using the `declarative_base` class from SQLAlchemy, the User table is created in a SQLite database.

Listing 9-9. SQLAlchemy setup

```

##SQLAlchemy Engine
from sqlalchemy import create_engine
from sqlalchemy.dialects.sqlite import *

SQLALCHEMY_DATABASE_URL = "sqlite:///./mydata.sqlite3"
engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_
    thread": False})

#Session object
from sqlalchemy.orm import sessionmaker, Session

session = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

##Users model
from sqlalchemy import Column, Integer, String

class Users(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True, nullable=False)
    username = Column(String(50), unique=True)
    password = Column(String(200))
    token = Column(String(200))

Base.metadata.create_all(bind=engine)

```

Note that the Users model has a token attribute that stores the hashed value of password. Let us provide a POST operation function `add_user()` to compute the token field and add a new user in the database (Listing 9-10).

Listing 9-10. POST operation function

```
@app.post('/users', response_model=User)
def add_user(u1: User, db: Session = Depends(get_db)):
    u1.token = hash(u1.password)
    u1.password=u1.password.get_secret_value()
    usrORM=Users(**u1.dict())
    db.add(usrORM)
    db.commit()
    db.refresh(usrORM)
    return u1
```

The preceding function has a database session dependency, which we had discussed in Chapter 7. From within the Swagger UI page, add admin as a new user.

We now have to include a GET operation on the **/users** route to generate the list of users (Listing 9-11). Further, we shall secure the `get_users()` operation function by injecting the authorization token in it.

Listing 9-11. Secured GET operation function

```
@app.get('/users', response_model=List[User])
def get_users(db: Session = Depends(get_db), token: str =
Depends(scheme)):
    recs = db.query(Users).all()
    return recs
```

If the dependencies are solved, the list of users in the database will be displayed. However, we need to modify the `token()` function (in Listing 9-12) to check the username and password entered by the user in the Password request form.

Listing 9-12. Authentication function

```
@app.post('/token')
async def token(form_data: OAuth2PasswordRequestForm =
Depends(), db: Session = Depends(get_db)):
    u1= db.query(Users).filter(Users.username == form_data.
    username).first()
    if u1.password == form_data.password:
        return {'access_token' : u1.token }
```

Make sure that all the preceding changes are made, and the server is running. From the Swagger UI page, authorize the `get_users()` function. The server's response will appear as in Figure 9-11.

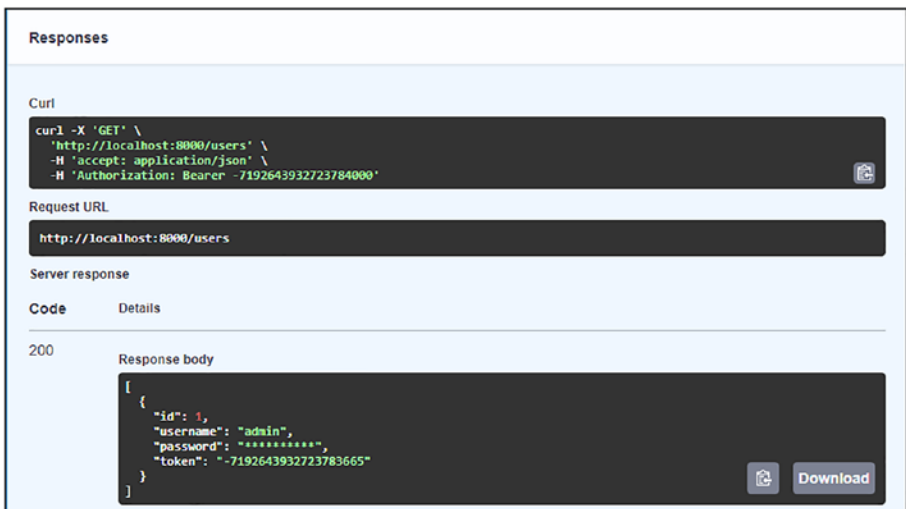


Figure 9-11. Access token in the response body

Note that the access token obtained is included as the Authorization header in the HTTP request.

Testing

FastAPI's testing functionality is based on the **HTTPX** client library. The `TestClient` object can issue requests to the ASGI application. You can then write useful unit tests and verify their results with **PyTest**.

As a prerequisite to writing and running unit tests, you need to install two libraries – HTTPX and PyTest – with the following commands:

```
pip3 install httpx
pip3 install pytest
```

Let us first have two path operations – a GET operation and a POST operation in the `main.py` script (Listing 9-13). The `list()` function retrieves an item from the Books list. The `addnew()` function is decorated with `@app.post()` and adds a book in the list. The `main.py` script is fairly straightforward.

Listing 9-13. GET and POST operation functions for testing

```
from fastapi import FastAPI
from pydantic import BaseModel

class Book(BaseModel):
    title: str
    price: int

books=[{"title":"Python", "price":500}, {"title":"FastAPI",
"price":750}]

app=FastAPI()

@app.get("/list/{id}")
async def list(id:int):
    return books[id-1]
```

```
@app.post("/list", status_code=201)
async def add_new(b1:Book):
    books.append(b1.dict())
    return b1
```

One notable thing about the preceding code, especially the POST decorator, is that a **201** status code is passed to it to imply that a successful POST operation creates a new resource.

It may be remembered that tests are saved in Python scripts whose name starts with **test_**. The test function should also be named as **test_***.

We shall write the tests in a `test_main.py` file. It is placed in the same folder in which the Python script with the FastAPI app object is declared. The folder must have an empty **`__init__.py`** file for the folder to be recognized as a package:

```
C:\fastapi\testing
|   main.py
|   test_main.py
|   __init__.py
```

To write a test, we should have a `TestClient` object and pass the FastAPI app object to it as the parameter (Listing 9-14).

Listing 9-14. `TestClient`

```
from fastapi.testclient import TestClient
from fastapi import status
from .main import app

client = TestClient(app)
```

In Listing 9-15, the `test_list()` function is intended to test the `list()` function. First, obtain the response of the **`/list/1`** URL with the `client.get()` method. Use the `assert` keyword to check if the status code is 200, which is what a successful response should normally have. We shall check if the

JSONized response equals the first item in the list (first because we are passing **1** as the path parameter). Add the `test_list()` function as in Listing 9-15.

Listing 9-15. Test GET operation

```
def test_list():
    response = client.get("/list/1")
    assert response.status_code == status.HTTP_200_OK
    assert response.json() == {"title": "Python", "price": 500}
```

Let us add another test function to check if the POST operation is carried out correctly by the `add_new()` function. The code for the `test_add_new()` function is given in Listing 9-16. For the POST call, we need to provide the JSON data as one of the parameters. As before, check the status code of the response, which is expected to be **201** (it stands for a new resource created).

Listing 9-16. Test POST operation

```
def test_add_new():
    response = client.post("/list", json={"title": "Learn
FastAPI", "price": 1000})
    assert response.status_code == status.HTTP_201_CREATED
```

Run the tests from the command line. PyTest automatically discovers the tests and tells if they pass or fail:

```
(fastenv) C:\fastenv\testing>pytest
===== test session starts =====
platform win32 -- Python 3.10.7, pytest-7.2.0, pluggy-1.0.0
rootdir: C:\fastenv\testing
plugins: anyio-3.6.2
```

```
collected 2 items
```

```
test_main.
```

```
py..
```

```
[100%]
```

```
===== 2 passed in 0.46s =====
```

Testing WebSocket

The `TestClient` object is also capable of sending a connection request to a WebSocket set up by a FastAPI endpoint. On the server side, the request is accepted, and a JSON message is sent to the client – in this case, the test function. The FastAPI code is very simple (Listing 9-17) – just refer to our earlier discussion about the WebSockets module in Chapter 8.

Listing 9-17. WebSocket with FastAPI

```
from fastapi import FastAPI

from fastapi.websockets import WebSocket

app = FastAPI()

@app.websocket("/wstest")
async def wstest(websocket: WebSocket):
    await websocket.accept()
    await websocket.send_json({"msg": "From WebSocket Server"})
    await websocket.close()
```

As in the previous topic, create an empty `__init__.py` file, and save the code shown in Listing 9-18 in `test_main.py` which should be alongside the `main.py` script.

Listing 9-18. WebSocket test function

```

from fastapi.testclient import TestClient
from .main import app

def test_wstest():
    client = TestClient(app)
    with client.websocket_connect("/wstest") as websocket:
        data = websocket.receive_json()
        assert data == {"msg": "WebSocket Server"}

```

As you can see, the `TestClient` object calls the `websocket_connect()` method to send a connection request. The `assert` statement compares the JSON data received from the server with the anticipated data. This time, we expect the test to fail. Here is the console output of the **PyTest** command:

```

===== FAILURES =====
_____ test_wstest _____

def test_wstest():
    client = TestClient(app)
    with client.websocket_connect("/wstest") as websocket:
        data = websocket.receive_json()
>       assert data == {"msg": "WebSocket Server"}
E       AssertionError: assert {'msg': 'From...ocket
Server'} == {'msg': 'WebSocket Server'}
E       Differing items:
E       {'msg': 'From WebSocket Server'} != {'msg':
'WebSocket Server'}
E       Use -v to get more diff

test_main.py:7: AssertionError
===== short test summary info =====

```

```

FAILED test_main.py::test_wstest - AssertionError: assert
{'msg': 'From...ocket Server'} == {'msg': 'WebSocket Server'}
===== 1 failed in 0.48s =====

```

Testing Databases

The operation functions underneath the API endpoints primarily perform CRUD operations on the back-end database. Hence, your tests should assert their satisfactory execution.

Often, you would like to set up a different database for testing rather than using the live database. For this purpose, you need to override the dependency that injects the session object with which you carry out the CRUD operations.

Override Dependency

To understand what is overriding of dependency and how it works, let us revisit the example in the section “Query Parameters As Dependencies” in Chapter 7. Here, the query parameters are injected by the `properties()` function. The relevant part of the code is reproduced here:

```

async def properties(x: int, y: int):
    return {"from": x, "to": y}

@app.get("/persons/")
async def get_persons(params: dict = Depends(properties)):
    return persons[params['from']:params['to']]

```

To write and run a unit test, we would rather like to define a new dependency and override the application’s dependency. The `dependency_overrides` property of the FastAPI application object allows you to do this. Put the function in the **test_main.py** file (Listing 9-19).

Listing 9-19. Overriding dependency

```

from fastapi import Depends
from fastapi.testclient import TestClient
from main import app, properties

async def new_properties(x: int=0, y: int=1):
    return {"from": x, "to": y}

app.dependency_overrides[properties] = new_properties

```

With this new dependency, you expect the persons list to contain only the first entry. We can now assert if it really is the case, with the test function in Listing 9-20.

Listing 9-20. Testing dependency

```

client = TestClient(app)

def test_overridden_depends():
    response = client.get("/persons/")
    assert response.status_code == 200
    assert response.json() == [
        {
            "name": "Tom",
            "age": 20
        }
    ]

```

Override get_db()

While discussing how FastAPI interacts with different databases (Chapter 6), we have frequently used the `get_db()` function as dependency and inject database session reference in the operation functions that perform CRUD operations.

For convenience, some important parts of the code in the SQLAlchemy topic are reproduced here. Listing 9-21 has Pydantic and SQLAlchemy models for the Book table, a database dependency injection function, and a POST operation function to add a new book.

Listing 9-21. POST operation on the Book table

```
class Books(Base):
    __tablename__ = 'book'
    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String(50), unique=True)
    price = Column(Integer)

Base.metadata.create_all(bind=engine)

from pydantic import BaseModel

class Book(BaseModel):
    id: int
    title: str
    price: int

    class Config:
        orm_mode = True

app=FastAPI()

def get_db():
    db = session()
    try:
        yield db
    finally:
        db.close()

@app.post('/books', response_model=Book)
def add_book(b1: Book, db: Session = Depends(get_db)):
```



```

bkORM=Books(**b1.dict())
db.add(bkORM)
db.commit()
db.refresh(bkORM)
return b1

```

Importing libraries, setting up the SQLite database **mydata.sqlite3**, declaring the Session object and other operation functions can be found in the source code repository of this book.

Let us concentrate on the `get_db()` function that injects the Session object into the `add_book()` function. As mentioned earlier, we would not be using this database (`mydata.sqlite3`) for running the tests. Instead, we need to configure another database **test.sqlite3** and inject its session object. In the **test_main.py** file (Listing 9-22), the function that yields the session object of the alternate database is defined.

Listing 9-22. Setting up the test database

```

SQLALCHEMY_DATABASE_URL = "sqlite:///./test.sqlite3"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_
    thread": False}
)
TestingSession = sessionmaker(autocommit=False,
autoflush=False, bind=engine)

Base.metadata.create_all(bind=engine)

def test_get_db():
    try:
        db = TestingSession()

```

```

        yield db
    finally:
        db.close()

```

We now override the `get_db` dependency with this newly defined dependency function – `test_get_db()`. It adds a test data in this `TestingSession` object, which is pointing to the test database (Listing 9-23).

Listing 9-23. Testing the `add_book()` function

```

app.dependency_overrides[get_db] = test_get_db

client = TestClient(app)

def test_add_book():
    response = client.post(
        "/books/",
        json={"id":1,"title": "Jungle Book", "price": 500},
    )
    assert response.status_code == 200
    data = response.json()
    assert data["title"] == "Jungle Book"
    assert "id" in data
    book_id = data["id"]

    response = client.get(f"/books/{book_id}")
    assert response.status_code == 200
    data = response.json()
    assert data["title"] == "Jungle Book"
    assert data["id"] == book_id

```

When you run this test, a POST request to the `/books` endpoint is initiated with the given data and returns the result of assertion if the retrieved instance equals the test data.

AsyncClient

All the tests in this topic so far have been synchronous in nature as the test functions are defined as normal Python functions and not coroutines (having an `async` prefix). That's because the `fastapi.TestClient` class doesn't support asynchronous calls. Thankfully, we have the **HTTPX** library whose `AsyncClient` class allows us to write async tests.

Asynchronous tests become especially important in an application that processes the back-end database asynchronously. Earlier in this book, we have learned how FastAPI handles asynchronous database operations.

For this section, we shall be using the FastAPI code in Listing 9-24. There is an in-memory database in the form of a list of books and asynchronous POST and GET operation functions.

Listing 9-24. FastAPI code with GET and POST operation functions

```
from fastapi import FastAPI
from pydantic import BaseModel

class Book(BaseModel):
    title: str
    price: int

books=[{"title":"Python", "price":500}, {"title":"FastAPI",
"price":750}]

app=FastAPI()

@app.get("/list/{id}")
async def list(id:int):
    return books[id-1]
```

```
@app.post("/list", status_code=201)
async def add_new(b1:Book):
    books.append(b1.dict())
    return b1
```

To communicate with the Uvicorn ASGI server, we'll use the `AsyncClient` object and then await the response from the API endpoint. You also must decorate the async test function with the `anyio` plugin of the `PyTest` library.

In Listing 9-25, the `AsyncClient` constructor takes two parameters – an `app` as the FastAPI application object and a `base_url`. We then assert if the status code in the response equals **200**.

Listing 9-25. Testing with `AsyncClient`

```
import pytest
from httpx import AsyncClient
from .main import app

@pytest.mark.anyio
async def test_list():
    async with AsyncClient(app=app, base_url="http://
localhost") as ac:
        response = await ac.get("/list/1")
        assert response.status_code == 200
        assert response.json() == {"title":"Python",
"price":500}
```

On the similar lines, you may add an asynchronous coroutine to test the POST operation. You can also try and write tests with `AsyncClient` for the FastAPI applications performing asynchronous database operations with **`aiosqlite`** (SQLite database) and the **`Motor`** driver (MongoDB database).

Summary

As mentioned in the beginning of this chapter, security and testing are important steps in the API development process. In this chapter, we learned a few techniques to provide a secure access to our API. We also learned how to write and run unit tests with the help of PyTest and HTTPX libraries.

In the next chapter, we'll get to know about different ways to deploy a FastAPI application.

CHAPTER 10

Deployment

The journey of web application development culminates when you make it publicly available for users. Once you are confident that your app is production-ready, it is time to explore the various options for its deployment.

In this chapter, we shall discuss the following topics related to the deployment of a FastAPI app:

- Hypercorn
- Daphne
- Gunicorn
- Render cloud
- Docker
- Google Cloud Platform
- Deta cloud

For any web application (or API) to be publicly accessible, it must be put on a remote machine with a public IP address. A high-performance HTTP server must be installed on it to serve the application. In the case of FastAPI, you need an ASGI-compatible server program. We have been using Uvicorn throughout this book. However, there are other alternatives such as Hypercorn and Daphne.

Hypercorn

Uvicorn, the ASGI server, doesn't support the **HTTP/2** protocol. On the other hand, Hypercorn supports HTTP/2 and HTTP/1 specifications, in addition to WebSocket (over both HTTP/1 and HTTP/2). With the use of the aioquic library, there is also an experimental support for HTTP/3 specifications.

HTTP/2 offers better efficiency over HTTP/1 because of several improvements. First, it uses the binary transfer protocol. Again, it employs a multiplexing technique to send multiple streams of data at once over a single TCP connection. For example, if the client requests for an **index.html** file (which internally uses an image file, say **logo.png**, and a stylesheet **style.css**), all the three resources are sent over a single connection rather than three as is the case in HTTP/1.

Table 10-1 summarizes the difference between HTTP/1 and HTTP/2.

Table 10-1. *HTTP/1 vs. HTTP/2*

	HTTP/1.1	HTTP/2
Released	1997	2015
Transfer Protocol	Text	Binary
Multiplexing	No	Yes
Server Push	No	Yes
Headers	Uncompressed	Compressed

HTTP/2 also uses a better header compression technique, and by the server push mechanism, resources are sent to the client even before they are requested.

To install Hypercorn, use the PIP installer as always:

```
pip3 install hypercorn
```

The usage of Hypercorn to serve a FastAPI app is the same as Uvicorn:

hypercorn main:app --reload

The server response section of the Swagger UI docs page identifies the server as **hypercorn-h11**, which indicates an HTTP/1.1 protocol (Figure 10-1).



Figure 10-1. Hypercorn server with an HTTP/1 protocol

HTTPS

One of the important considerations while deploying an API is to ensure that the server accepts only secure requests from the client. Although HTTPS uses the same URI scheme as does HTTP, it indicates to the client browser that it should use an added encryption layer to protect the traffic.

For HTTPS, the server needs to have a certificate. We can create a self-signed certificate using the RSA cryptography algorithm.

Open the **Git Bash** terminal and enter the following command:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
privatekey.key -out certificate.pem
Generating a RSA private key
.....+++++.....+++++
writing new private key to 'privatekey.key'
-----
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
```

This will generate two files – **privatekey.key** and **certificate.pem**. We can now use these files as values for `keyfile` and `certfile` command-line options with `Uvicorn` and `Hypercorn`.

Run the `Uvicorn` server with the following command:

```
uvicorn main:app --ssl-keyfile=./privatekey.key --ssl-
certfile=./certificate.pem --reload
INFO:      Will watch for changes in these directories:
          ['C:\\fastenv\\certificate']
INFO:      Uvicorn running on https://127.0.0.1:8000 (Press
          CTRL+C to quit)
INFO:      Started reloader process [4100] using WatchFiles
INFO:      Started server process [13928]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Note that the URL where the app is running uses the HTTPS scheme.

Use the same key and certificate files with Hypercorn in the following command line to enable the HTTPS scheme:

```
hypercorn --keyfile privatekey.key --certfile certificate.pem  
main:app
```

Go back to the documentation page (<https://localhost:8000/docs>) and check the response headers (Figure 10-2).

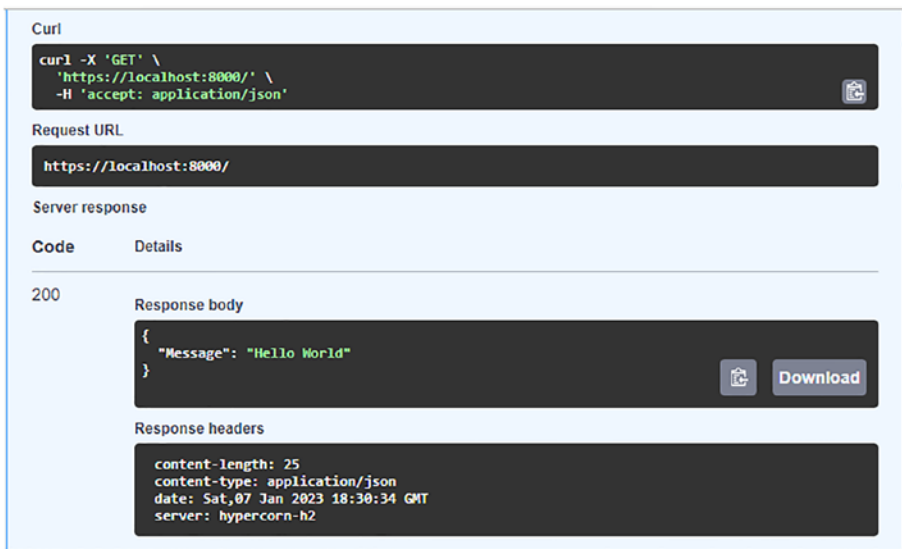


Figure 10-2. Hypercorn server with an HTTP/2 protocol

The name of the server as **hypercorn-h2** means that the HTTP/2 protocol is being implemented.

Daphne

Another ASGI implementation used widely for deploying a FastAPI app is **Daphne**. It was originally developed to power **Django Channels** – a wrapper around the Django framework to enable ASGI support.

Daphne also supports HTTP/2 and WebSocket protocols. While Uvicorn depends upon the `uvloop` library and Hypercorn on `trio`, Daphne implements async loops with a `twisted` library.

To launch the FastAPI app on Daphne with the SSL certificate, use the following syntax:

```
daphne -e ssl:8000:privateKey=privatekey.  
key:certKey=certificate.pem main:app
```

```
2023-01-08 10:19:16,339 INFO      Starting server at  
ssl:8000:privateKey=privatekey.key:certKey=certificate.pem  
2023-01-08 10:19:16,339 INFO      HTTP/2 support enabled  
2023-01-08 10:19:16,339 INFO      Configuring endpoint  
ssl:8000:privateKey=privatekey.key:certKey=certificate.pem
```

Note that the HTTP/2 support is automatically enabled.

Gunicorn

With Uvicorn, the FastAPI app is run as a single process. However, in the production environment, you would like to have some replication of processes. If there are more clients, the single process can't handle them even if the server's machine has multiple cores. It is possible to have multiple processes running at the same time and distribute the incoming requests among them. Such multiple processes of a single API are called workers.

Gunicorn is another HTTP application server. Although it is a WSGI-compliant server (which means, by itself, it is not compatible with FastAPI), its process manager class allows the user to choose which worker process class to use.

Uvicorn has a **Gunicorn-compatible worker** class. Gunicorn's process manager delegates the requests to the worker class for conversion of incoming data to the ASGI standard and further consumption by the FastAPI app.

First, install Gunicorn and then run the following command (note that Gunicorn doesn't support Windows OS):

```
gunicorn main:app --workers 4 --worker-class uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000
```

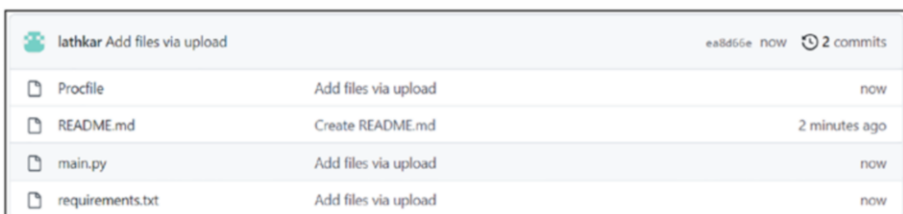
If you choose to use a cloud-based hosting service to deploy your API, it will handle the replication even if there's a single Uvicorn process.

FastAPI on Render Cloud

One of the easiest ways to deploy your API for public consumption is to use one of the many cloud-based services. Examples are Heroku, Google Cloud Platform, and more. Such services can be availed with paid subscription, although most of them do have a free tier.

In this section, we shall learn how to deploy a simple FastAPI app on the Render cloud (<https://render.com>) within minutes.

One of the unique features of Render is its ability to autodeploy the app from the GitHub repository. Hence, you need to create a repository on <https://github.com> and place the source code of your app along with the requirements.txt file in it, as in Figure 10-3.



File	Action	Time
Profile	Add files via upload	now
README.md	Create README.md	2 minutes ago
main.py	Add files via upload	now
requirements.txt	Add files via upload	now

Figure 10-3. FastAPI app source code in the GitHub repository

The next step is to sign up with <http://render.com> using your GitHub account (Figure 10-4).

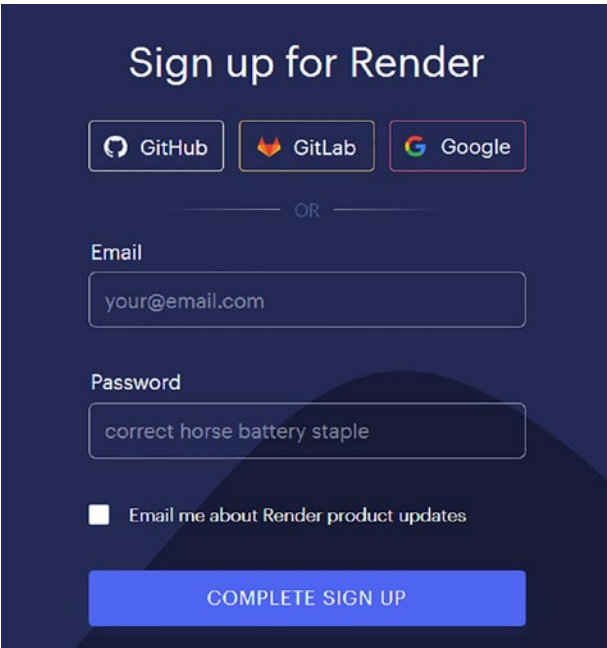


Figure 10-4. Sign up for Render

Next up, connect your GitHub repository so that Render fetches its contents (Figure 10-5).

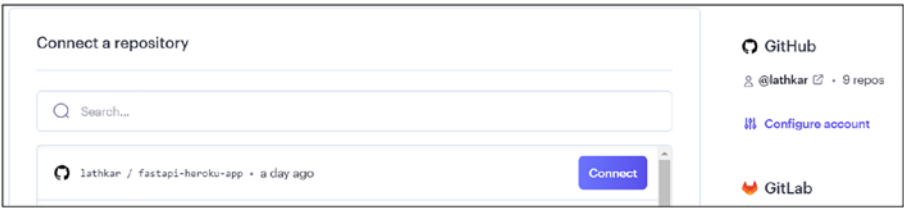


Figure 10-5. Connect the GitHub repository to Render

You now need to tell Render the following details:

Environment:	Python 3
The runtime environment for your web service	
Build command:	<code>pip install -r requirements.txt</code>
A script that installs the required libraries	
Start command:	<code>uvicorn main:app --host 0.0.0.0 --port 10000</code>

After the deployment is complete, the Render dashboard shows the public URL of your app (Figure 10-6).

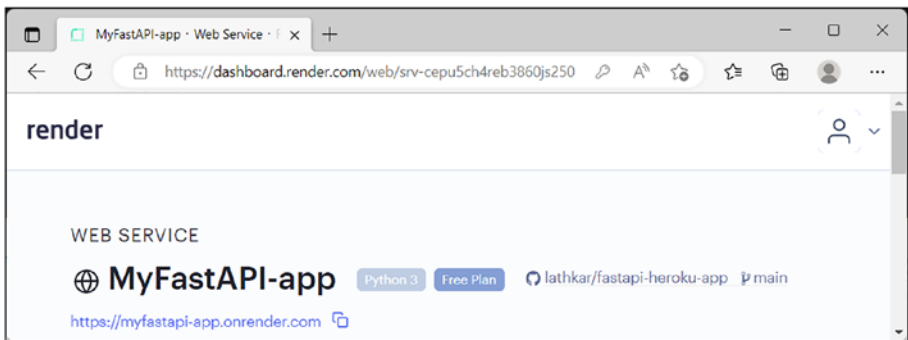


Figure 10-6. *Render dashboard*

The procedure of hosting an app on some of the other cloud platforms such as Heroku (<https://heroku.com>) and Cyclic (<https://cyclic.sh>) is more or less the same. (Heroku has since discontinued the free tier option!)

Docker

Instead of employing the preceding method for deployment, where the code is put on the cloud platform, and invoking the application server from there, developers prefer the approach of building a **container image** consisting of all the dependencies of the app. The container image is then deployed either on a server machine or on the cloud platform.

Containers are lightweight as compared to **virtual machines** (VMs). They are easily portable. You can build a container on your local machine and deploy it to any environment.

Docker is one of the most popular platforms for developing, distributing, and running applications. Building containers with Docker gives many advantages including security, replicability, simplicity, etc.

The Docker platform consists of Docker Engine that works on top of the host operating system, and more than one container can be built with it (Figure 10-7).

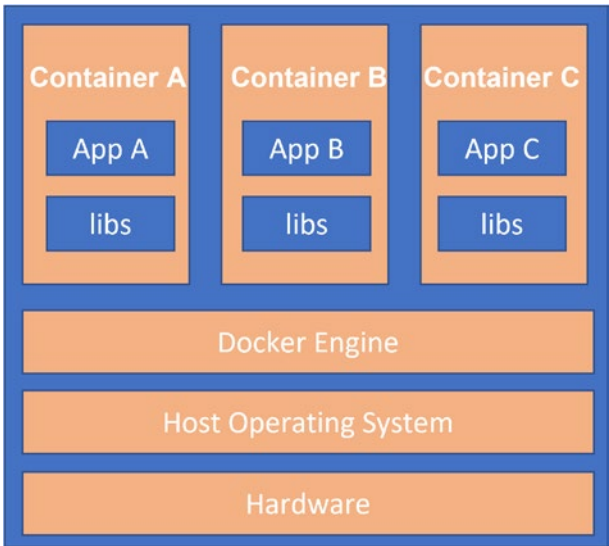


Figure 10-7. Docker containers

Let us now build a Docker image for a simple FastAPI application.

Keep your `main.py` file along with the `requirements.txt` file in a folder. Alongside it, save the following text as **Dockerfile** (this file doesn't have any extension):

```
FROM python:3.10
RUN pip3 install -r requirements.txt
copy ./app app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

We need to download and install the **Docker Desktop** application from www.docker.com/products/docker-desktop/.

Docker Engine uses this Dockerfile to construct the image. It basically tells the engine to use the official Python image as the base, install the dependencies, copy files from the source folder, and launch the application with Uvicorn.

Open the Git Bash terminal, navigate to the project directory where you have stored the Dockerfile, and run the following command:

```
$ docker build -t image1 .
```

To start an instance of the image just built, use the following command:

```
$ docker run -d --name mycontainer -p 80:80 image1
```

Your application is now running on the localhost.

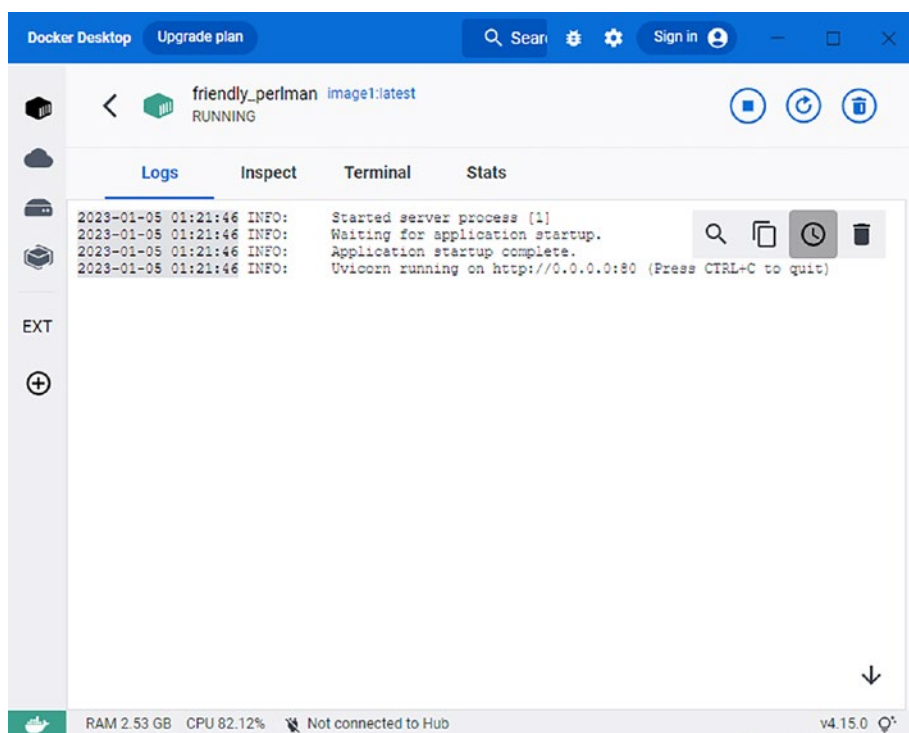


Figure 10-9. Running the Docker image

You can now deploy the Docker image on various platforms such as Heroku, AWS, Google Cloud, etc.

Google Cloud Platform

The Internet giant Google offers cloud computing services in the name **Google Cloud Platform** (GCP). Web applications in various language technologies including Python can be deployed on Google Cloud.

A FastAPI app can be hosted on Google Cloud in either of two ways. First is using the Docker image and deploying it. Second is to host the application code on GCP along with its dependencies and serve the application on the public IP address provided by Google Cloud. We shall briefly discuss the second approach.

The first step is to put the FastAPI app code in a GitHub repository, as we had done while deploying on Render. Be sure to include the `requirements.txt` file in the repository.

Just as you need a Dockerfile to build the Docker image, GCP needs a text file named **app.yaml** in the repository.

YAML is a human-readable data serialization language. Various application settings such as runtime and version numbers are placed in it.

Save the following script as `app.yaml` and put it in the GitHub repository:

```
runtime: python37
entrypoint: gunicorn -w 4 -k uvicorn.workers.UvicornWorker
main:app
```

To host your application on Google Cloud, obviously you need to sign up and log in to <https://console.cloud.google.com/>. (Google Cloud lets you host your app for free, but you should provide billing information including bank details.)

After logging in, go to <https://console.cloud.google.com/cloud-resource-manager> and create a new project (Figure 10-10). Choose the name of the project and the project ID.

Google Cloud

New Project

Project name *
fastapi-app-test

Project ID *
fastapi-app-test


Project ID can have lowercase letters, digits or hyphens. It must start with a lowercase letter and end with a letter or number.

Location *
 No organisation [BROWSE](#)

Parent organisation or folder

[CREATE](#) [CANCEL](#)

Figure 10-10. *New project in GCP*

To perform the rest of the activity, open the Cloud Shell by clicking the  button, found toward the right side of the Google Cloud dashboard.

The next step is to clone the GitHub repository (in which you have put the application code, **requirements.txt** and **app.yaml** files). Here is an example (obviously, you will use the URL of your GitHub repository here):

```
git clone https://github.com/lathkar/fastapi.git
```

Initially, you run the application as if you have been running so far on your local machine. To do so, create the virtual environment, install the libraries from **requirements.txt**, and run the application.

Execute the following commands from within the Cloud Shell itself:

```
fastapi (fastapi-app-test)$ cd fastapi
fastapi (fastapi-app-test)$ virtualenv venv
```

CHAPTER 10 DEPLOYMENT

```
fastapi (fastapi-app-test)$ source env/bin/activate
fastapi (fastapi-app-test)$ pip install -r requirements.txt
fastapi (fastapi-app-test)$ gunicorn -w 4 -k uvicorn.workers.
UvicornWorker main:app
```

The log emitted on the console should be familiar as you have seen it while running the code examples in this book:

```
[2023-01-09 03:48:22 +0000] [587] [INFO] Starting
gunicorn 20.1.0
[2023-01-09 03:48:22 +0000] [587] [INFO] Listening at:
http://127.0.0.1:8000 (587)
[2023-01-09 03:48:22 +0000] [587] [INFO] Using worker: uvicorn.
workers.UvicornWorker
```

The application is now running on the localhost, which, if you wish, you can verify by visiting the URL.

Now we have to deploy this application, so that it is publicly accessible. To do that, first create the app:

```
$ gcloud config set project VALUE
$ gcloud app create
```

Replace VALUE by the project ID generated at the time of creating the project. You will be asked to choose the region code. The recommended way is to choose the region closest to your geographical location, so that the response time of the app is optimized:

```
Creating App Engine application in project [fastapi-app-test]
and region [asia-southeast2]....done.
Success! The app is now created. Please use `gcloud app deploy`
to deploy your first app.
```

The final step is to deploy the project. Google Cloud uses the YAML file available in the project directory. It will display the target URL before asking for confirmation:

Services to deploy:

```
descriptor:          [/home/mlathkar/fastapi/app.yaml]
source:              [/home/mlathkar/fastapi]
target project:      [fastapi-app-test]
target service:      [default]
target version:      [20230109t041024]
target url:          [https://fastapi-app-test.et.r
                    .appspot.com]
target service account: [App Engine default service account]
Do you want to continue? (y/n)
```

The deployment procedure will take a while for completion. After successful completion, visit the target URL to verify successful deployment.

Deta Cloud

In the last section of this chapter, we shall introduce another cloud hosting service, **Deta** (<https://deta.sh>). Importantly, hosting your app on Deta is free (at least for now, although it claims that it will be forever!). Incidentally, Deta is one of the sponsors of FastAPI.

Deta offers the following products:

Deta Base is a fast, scalable, and secure NoSQL database. It can be used in serverless applications, while prototyping an application, in stateful integrations, and more.

Deta Micros (short for microservers) are web apps running at a specific HTTP endpoint. At the moment, you can build apps based on Node JS and Python.

Deta Drive is a file hosting service, with 10GB storage limit per Deta account. For instance, you may want to build an image server with the FastAPI/Python app as the front end and Deta Drive to store the images.

As always, you must sign up on the Deta platform with a username and password of your choice along with a user email which you need to verify to use the services.

Make sure that you have an error-free code for a FastAPI app in a folder which has the Python virtual environment installed. Also, save a `requirements.txt` file in the same folder.

To work with the Deta products, you must install the **Deta CLI** (command-line interface), as it is through this CLI that all the operations are done.

You can install the Deta CLI on Mac, Linux, and Windows. For Linux/Mac, execute the following command from the terminal:

```
$ curl -fsSL https://get.deta.dev/cli.sh | sh
```

For installation of the Deta CLI on Windows, open the PowerShell terminal, and issue the following command from within the folder which contains the `main.py` and `requirements.txt` files:

```
PS C:\fastapi-app>iwr https://get.deta.dev/cli.ps1 -useb | iex
```

This command downloads and installs the binary and adds deta in the system path.

After successful installation, run the **deta login** command from the PowerShell/Linux terminal. You will be redirected to the Deta sign-in page. After your account credentials are verified, the control comes back to the terminal.

```
PS C:\fastapi-app> deta login
Please, log in from the web page. Waiting...
https://web.deta.sh/cli/64048
Logged in successfully.
```

The **deta new** command builds the micro. Since ours is a Python app, specify it in the command line along with the name of the app:

```
PS C:\fastapi-app> deta new fastapi-app --python
Successfully created a new micro
{
    "name": "fastapi-app",
    "id": "59a94e9b-ec10-47a3-8623-a78adeb1f291",
    "project": "d0cuictl",
    "runtime": "python3.9",
    "endpoint": "https://vprjh5.deta.dev",
    "region": "ap-south-1",
    "visor": "disabled",
    "http_auth": "disabled"
}
Adding dependencies...
Collecting fastapi
...
Successfully installed anyio-3.6.1 fastapi-0.85.0 idna-3.4
pydantic-1.10.2 sniffio-1.3.0 starlette-0.20.4 typing-
extensions-4.3.0
```

Your app is now available at the endpoint URL. Visit the same in your browser to obtain a response from the app.

By default, the HTTP authentication is disabled. To enable it, generate the **access token** from your account's settings (Figure 10-11). The token is valid for 365 days.

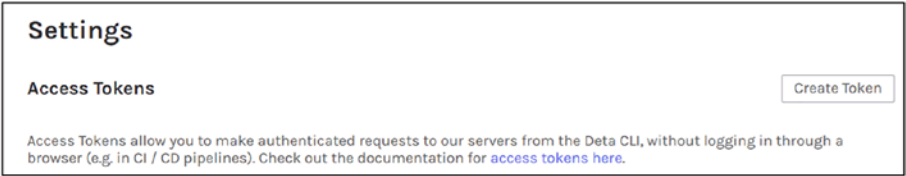


Figure 10-11. Generate the access token

Export the value of the token as the **ACCESS_TOKEN** environment variable. This enables sending authenticated requests from the Deta CLI so that you don’t need to log in from the CLI every time.

Go to the Deta dashboard. You will see the details of your app that has just been deployed, as in Figure 10-12.

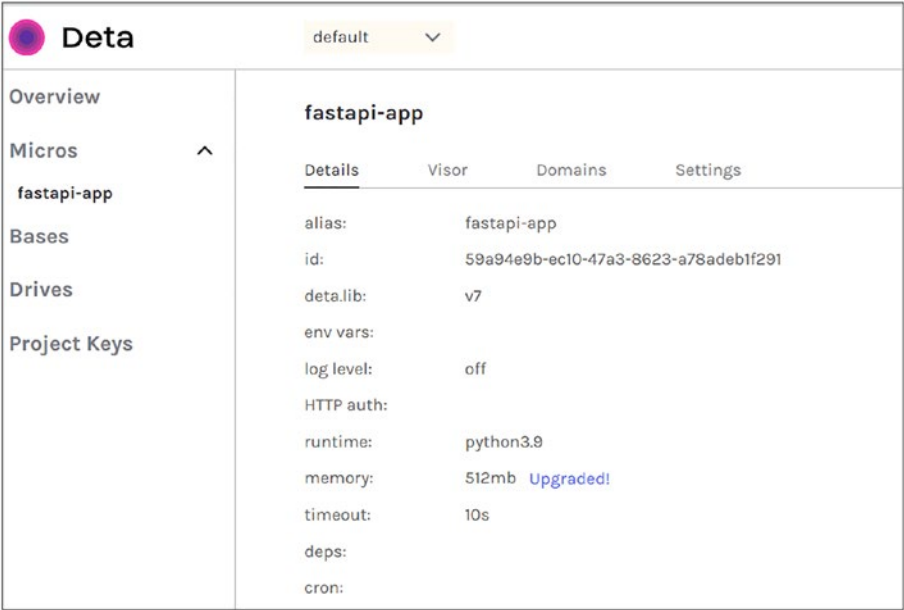


Figure 10-12. Deta dashboard

Several other cloud platforms can also be used to deploy a FastAPI app. They follow more or less the same procedure as we have seen in this chapter. Hence, to avoid repetition, they haven't been explained.

Summary

This is the last chapter of this book. In this chapter, you learned how to use different tools to deploy your FastAPI app. Alternative ASGI servers have been explained. The procedure of deploying the app on Render, GCP, and the Deta cloud has been described in a simple and step-by-step manner. Following these, you should be easily able to deploy your own app.

Index

A

Accept-language header, 130, 131

Access token, 250, 251, 259,
291, 292

add_book() function, 148, 155, 160,
167, 174, 269

add_header() function, 207

addnew() function, 67–69,
73–75, 83, 260

aiosqlite module, 154–156, 164,
173, 179

albums/albums.py module, 192

albums.py script, 186, 187

anyio, 25

AnyUrl type, 82

API documentation, 8, 36, 42,
54, 57, 62

API documentation tools, 36, 64

APIRouter

- albums module, 190

- albums.py script, 186, 187

- albums routes, 189

- books API routes, 188

- books router, 186

- fastapi module, 185

- individual, 185

- include_router() method, 187

- main app code, 191

- main.py script, 187, 188

- REST operations, 185

- router package, 189–191

- @app.delete(), 31, 163

- @app.get() decorator, 31, 63, 95, 98,
140, 161

- Application object, 30–33, 46, 185,
187, 191, 233, 240, 265, 271

- Application Programming Interface
(API), 248

- analogy, 14

- company's application, 15

- electromechanical peripheral
devices, 14

- social media services, 16

- stand-alone application, 14

- user registration and
authentication, 15

- “application/schema+json”, 36, 44

- @app.post() decorator, 31, 67, 118,
126, 160

- @app.put(), 31, 162

- @app.websocket() decorator, 224

- ASCII characters, 66

- asdict() and astuple() methods, 71

- asgiref package, 23

INDEX

- asgiref.server class, 23
- async/await mechanism, 9
- AsyncClient, 270, 271
- Asynchronous capabilities, 8, 11, 12
- Asynchronous dependency
 - function, 155, 166
- Asynchronous fetch methods, 154
- Asynchronous processing
 - ASGI, 11–13
 - asyncio module, 8–10
 - cooperative multitasking, 8
 - event/condition, 8
 - Python 3.5, 8
- Asynchronous Server Gateway
 - Interface (ASGI), 8, 11
 - application, 13
 - application coroutine, 11
 - callable coroutine, 11
 - callable object, 11
 - server, 67
- Asynchronous tests, 270
- async in SQLAlchemy
 - asynchronous dependency
 - function, 166
 - Core Expression Language,
 - 165, 166
 - databases module, 164
 - engine object, 165
 - FastAPI path
 - operations, 167–170
 - GET operation, 169
 - metadata object, 165
 - PUT operation, 169
 - SQLAlchemy Core, 163

- table class methods, 166, 167
- table object, 165
- asyncio module, 23
 - async keyword, 9
 - await keyword, 9
 - coroutine, 9
 - Python’s ASGI, 8
 - Python script, 10
 - run() function, 9
- Authentication function, 259
- await keyword, 9, 168, 179

B

- BaseConfig class, 76
- BaseModel class, 70, 71, 73
- Pydantic’s BaseModel class,
 - 71, 79, 80
- Bearer token, 250, 252
- Body class constructor, 68, 69
- Body() functions, 81
- Body parameter, 67–69, 123, 147,
 - 152, 160, 162, 168, 169,
 - 194, 255
- branch_id value, 61
- broadcast() method, 223

C

- Cacheability, 18
- __call__() method, 203
- call_next function, 207
- Client-server model, 17
- Code on demand, 18

- colorama, 25
 - Command Line Interface
 - Creation Kit, 25
 - Conditional expressions, 104
 - Config attribute, 76
 - connect() function, 144
 - Connection_handler class, 223, 224
 - Cookies
 - definition, 125
 - mechanism, 125
 - parameters, 126–128
 - set_cookie() method, 125, 126
 - tracking, 125
 - Core Expression Language,
 - 165, 166
 - Coroutine, 9–11, 47, 136, 138,
 - 154–156, 164, 166–169, 177,
 - 178, 196, 197, 213–216, 219,
 - 224, 270, 271
 - CORS, *see* Cross-Origin Resource
 - Sharing (CORS)
 - CORSMiddleware, 209
 - create_all() method, 166
 - create_engine() function, 165
 - CREATE, READ, UPDATE, and
 - DELETE operations, 19
 - credentials() function, 200, 201
 - Cross-Origin Resource Sharing
 - (CORS), 209, 210
 - CRUD operations, 19, 20, 70, 143,
 - 144, 156, 157, 166, 181, 195,
 - 265, 266
 - CSS files, 113
 - Curl command-line tool, 66
 - /customer route, 89
 - Customers model, 88
 - Custom middleware, 207
 - Custom validation, 85, 86
 - Cython-based
 - dependencies, 23, 24
- ## D
- d option, 66
 - Daphne, 24, 273, 277, 278
 - Databases module, 163, 164, 179
 - Database session
 - dependency, 204, 205, 258
 - @dataclass decorator, 70, 71
 - dataclasses library, 70
 - Dataclasses module, 70–71
 - data.get('name'), 102
 - data.get('salary'), 102
 - Data model with pydantic, 69–75
 - DB-API
 - create books table, 144–146
 - cursor object, 145
 - database driver, 144
 - database operations, 144
 - delete book, 153, 154
 - insert new book, 147–150
 - List of books, 150, 151
 - POST operation–SQLite, 148
 - Python code, 144
 - sample books data, 150
 - select all books, 150
 - select single book, 151, 152
 - SQLite, 144, 146

INDEX

DB-API (*cont.*)

- swagger UI, POST
operation, 149

- update book, 152, 153

default_response_class, 140

del_book() function, 153, 156, 163

DELETE method, 21

DELETE request, 21

Dependencies

- callable class, 203

- callable object, 203

- class, 202, 203

- context manager, 205

- database session, 204, 205

- decorator, 205, 206

- Depends(), 196

- function, 148, 159, 196

- generator function, 204

- get_db() function, 195

- global, 206

- mechanism, 195

- parameterized dependency

 - function, 200, 201

- path operation function, 195

- query parameters, 196–200

Depends() function, 147, 195,
202, 203

Deta Base, 289

Deta CLI, 290, 292

Deta Cloud

- access token, 292

- dashboard, 292

- Deta Base, 289

- Deta CLI, 290

Deta Drive, 290

deta login command, 290

Deta Micros, 289

deta new command, 291

FastAPI, 289

Deta Drive, 290

Deta Micros, 289

dict object, 32, 76, 100, 101, 126

dict() method, 79

division() function, 3–5

Docker

- advantages, 282

- container image, 282

- containers, 282

- desktop, 284

- engine, 282, 283

- run Docker image, 285

- VMs, 282

Dynamic typing, 2

E

delete() method, 163, 167, 169

EmailStr, 82

email-validator module, 82

Employee model, 82, 85

__eq__() function, 71

Event handlers, 22, 219

Exception handling

- get_name() function, 244

- HTTP error response, 245

- HTTPException, 244–246, 250

- HTTP protocol, 244

- runtime error, 243

user-defined exception,
245, 246

execute() method, 145, 154

F

fastapi module, 126

FastAPI, 210

- application, 28, 141
- ASGI-compliant framework, 13
- asynchronous processing, 8–13
- asynchronous web
 - framework, 21
- class, 31
- capabilities, 1
- dependency libraries, 28
- events, 238, 239
- file, 108
- GitHub repository, 279
- “Hello World” app, 26
- Hello World message, 30–36
- interactive API docs, 36–45
- object, 98
- operation function, 121
- OpenAPI, 22
- path operation function,
 - 167–170, 194
- PIP utility, 24
- Render Cloud, 279–281
- Sebastian Ramirez, 24
- type hints, 2–6
- web application
 - framework, 24, 29
- WebSockets, 215, 216, 218, 220

FastAPI dependencies

- libraries, 22
- Pydantic, 23
- Starlette, 22, 23
- Uvicorn, 23, 24

fastapi.Form class, 117

fastapi.responses module, 95

fastapi.templating module, 98

fetchone() method, 152

Field() function, 81

FileResponse, 94, 138, 139

FileResponse object, 138

find() method, 176, 179

Flask subapplication, 240

/form URL path, 117

format() function, 95

freeze subcommand, 25

from_orm() method, 79

Function *vs.* coroutine, 9

G

get_book() coroutine, 168

get_book() function, 150, 156, 161,
176, 179, 204

get_collection() function, 177, 178

get_cursor() function, 147, 155

get_db() dependency function, 204

get_db() function, 195, 266, 268

get_employee() function, 47–49,
55, 56, 59, 61, 199

getform() function, 118

get() method, 102

GET method, 20, 66

INDEX

- get_name() function, 244
- GET operation, 20, 150, 152, 162, 163, 168, 169, 176–179, 200, 204, 258, 260, 262, 270
- get_persons() operation
 - function, 197
- GET request, 20, 32, 34, 65, 66, 168
- get_users() function, 258, 259
- GitHub repository, 279, 280, 286, 287
- Global dependency, 206
- Google Cloud Platform (GCP)
 - Cloud Shell, 287
 - Docker image, 285
 - FastAPI app, 285
 - FastAPI app code, GitHub repository, 286
 - GitHub repository, 287
 - host application code, 285
 - new project, 287
 - YAML, 286, 289
- Grant type, 251
- GraphQL
 - architecture, 227
 - Facebook, 226
 - feature, 227
 - mutations, 229, 230
 - queries, 228, 229
 - query language, APIs, 227
 - REST APIs, 226
 - schema, 231
 - SDL, 227, 228
 - Strawberry, 232–238
 - subscriptions, 230, 231

- type definition, 228
- Gunicorn, 278–279

H

- handleOnClick() function, 217
- Hard-coded HTML string, 98
- hash() function, 255
- /header URL, 130
- Header parameter, 130–131
- Headers
 - “Content-Language”, 129
 - HTTP header types, 129
 - response headers section, 130
 - set_header() function, 130
 - “X-Web-Framework”, 129
- hello() function, 224
- hello.html, 98
- Hello World application code, 32
- Hello World message program
 - application object creation, 30
 - externally visible server, 34–36
 - objective, 30
 - path operation decorator, 30–31
 - path operation function, 32
 - Uvicorn server, 32–34
- Hello world template, 98
- Hello World text, 100
- h11 package, 26
- HTML response, 93–96, 98, 135
- HTML’s H2 style, 94
- HTTP-based client-server
 - communication, 131
- HTTPBasic class, 248

HTTP DELETE method, 153
 HTTPException, 244–246, 250
 HTTP method, 19, 31, 66, 143
 HTTP protocol, 22, 23, 66, 211, 213, 244, 248
 HTTPS, 275–278
 HttpUrl, 82
 HTTP verbs
 DELETE method, 21
 GET method, 20
 POST verb, 19, 20
 PUT method, 20, 21
 Hypercorn, 24
 HTTP/1, 274, 275
 HTTP/1 *vs.* HTTP/2, 274
 HTTP/2, 274, 277
 HTTPS, 275–278
 hypercorn-h11, 275
 PIP installer, 274

I

Identical POST requests, 20
 include_in_schema property, 64
 include_router() method, 187
 index() function, 32, 34, 94, 127, 254
 index() operation function, 139
 index() view function, 100
 __init__() constructor, 70
 init_db() function, 146
 insert() method, 166
 insert_one() method, 174
 Insomnia, 221, 222, 235, 236

Interactive API docs
 JSON schema, 44, 45
 OAS, 36
 path parameters, 46–49
 query parameters, 49–55
 Redoc, 42–44
 Swagger UI, 36–42
 Interface, 14
 Inventory_val, 122
 IP address, 36
 ipconfig command, 35
 IPython, 3, 4
 isalnum() method, 85

J, K

JavaScript function, 109, 110
 jinja2
 conditional and loop statements, 120
 conditional logic, 103
 CSS, 115
 data.get('langs'), 106
 endif keyword, 103
 for and endfor keywords, 105
 JavaScript, 109
 language syntax, 100
 library, 97
 package, 97
 placeholders, 100, 101
 profile.html, 106, 114
 server-side templating library, 100
 static image, 112

INDEX

jinja2 (*cont.*)

- template engine, 107, 108
- template variables, 103
- template web page, 104
- url_for() function, 113

Jinja2Templates function, 98

JSON-based media, 36

JSON response, 102, 121, 136

JSON schema, 44, 45

JSONType class, 93

Jupyter Notebook, 3

L

Layered system, 18

list() function, 260

Logical expressions, 104

M

main(), 10

main.py, 12, 32, 187, 188

- max_anystr_length, 76

media_type parameter, 94, 121,
134, 135, 137, 138

Metadata properties, 62, 63

Middleware

- add_header() function, 207
- @app.middleware (“http”), 207
- call_next function, 207
- custom, 207
- GZipMiddleware, 208
- Header insert, 208
- HTTP, 207

HTTPSRedirectMiddleware, 208

TrustedHostMiddleware, 208

min_length, 81

ModelB class, 87

Modern web application

- frameworks, 31, 96

MongoDB

- Motor (*see* Motor)

- PyMongo (*see* PyMongo)

MongoDB Compass, 171–173

MongoDB Query Language, 171

Motor

- client class, 177

- get_collection() function,
177, 178

- GET operation, 178, 179

- POST operation decorator, 178

- PyMongo, 177

mount() function, 107, 191

Mounting Subapplications, 191,
193, 194

Mounting WSGI

- application, 239–241

Multiclient Chat

- Application, 222–226

Mutations, 229, 230, 237

myfunction(), 9

myFunction, 109

mystyle.css file, 114

N

name parameter, 63

Nested models, 87

Numeric parameters
validation, 60, 61

O

OAuth, 251–253
access token, 250
bearer token, 250
features, 250
grant type, 251
Open Authorization, 250
password flow, 252
Pydantic model, 251
SQLAlchemy model, 251
OAuth2PasswordBearer
access authorization, 256
access token, response
body, 259
authentication function, 259
fastapi.security module, 252
GET operation function, 258
index() function, 253
password request form, 254
POST operation function, 258
server response with 401
code, 254
SQLAlchemy setup, 257
token() function, 255, 258
Users model, 257
OAuth2PasswordRequestForm, 255
Object langs, 105
Object-relational mappers
(ORMs), 77, 156
object, 156

relation, 156
open() function, 138
OpenAPI
standards, 31
path operation, 122
OpenAPI Specification (OAS), 36
Operation function, 115
Optional parameters, 51–53
Order of parameter
declaration, 54, 55
ORJSONResponse, 136
orm_mode, 77–79

P

Parameterized dependency
function, 200, 201
Parameter substitution, 95, 101
Parameters validation
fastapi module, 56
metadata, 62–64
numeric parameters, 60, 61
optional keyword
parameters, 56
RegEx, 59
standard Python types, 56
string parameter, 56–58
user inputs, 55
Path and Query
constructors, 69
Path() function, 59, 81
Path/endpoint, 31
Path operation decorator, 30–31,
37, 67, 185

INDEX

- Path operation function, 32, 41,
 - 156, 159, 160, 166, 174, 178,
 - 188, 194–196, 201, 207, 216,
 - 240, 246, 248, 252
- Path parameters
 - decorator, 46
 - employee, 46
 - handler function, 46
 - IP address:port, 46
 - operation decorator, 46
 - request URL, 46
 - type hints, 47
 - type parsing, 48, 49
- pip3 install jinja2
 - command, 97
- Placeholder identifier, 62, 67
- PlainTextResponse, 94
- POST Curl command, 68
- Postman app, 66
- POST method, 19, 20, 66, 68, 74
- POST operation function, 31, 67,
 - 74, 80, 147
- POST request, 20, 21, 66, 67, 88,
 - 117, 122, 269
- prod_alchemy, 79
- Product objects, 74
- ProductORM model, 79
- Products model, 87–89
- ProductVal model, 122, 123
- profile.html, 114
- PUT method, 20, 21
- PUT request, 21
- Pydantic library, 70, 71, 82
- Pydantic fields, 80, 81, 91
- Pydantic models, 21, 23, 64, 87, 91,
 - 101, 121, 122, 147, 148,
 - 159–161, 174, 251
- parameter, 73–75
- structure, 78
- Pydantic’s built-in validation, 84
- Pydantic types, 83
- PyMongo
 - add_book() function, 174
 - @app.get() decorator, 176
 - BSON representation, 175
 - Collection object, 174
 - document, 170
 - get_book() function, 176
 - GET /books request, 176
 - GET operation, 176, 177
 - insert_one() method, 174
 - localhost, 171
 - MongoClient class, 173
 - MongoDB Compass, 171–173
 - MongoDB Query Language, 171
 - MongoDB shell, 171
 - PIP utility, 173
 - POST operation, 174
 - Pydantic model, 174
 - schemaless, 170
 - start MongoDB server, 170
- Python, 213, 231, 232
 - application frameworks, 13
 - built-in data types, 80
 - dictionary object, 102
 - dynamic typing, 2
 - function, 3
 - IDEs, 3, 4

- interaction, 2
- interpreter, 3, 4
- main.py, 33
- modern features, 24
- multithreading approach, 8
- prompt, 3
- standard library, 12
- variable, 2
- VS Code, 4
- wsgiref module, 12
- Python-based web apps, 97
- python-dotenv, 26
- python-multipart package, 118

Q

- Queries, 228, 229
- Query() functions, 81
- Query parameters
 - @app.get() decorator, 49
 - get_employee() function, 49
 - optional parameters, 51–53
 - order of, 53–55
 - path decorator, 49
 - path parameter, 49
 - query string, 51
- Query's payload, 229

R

- readfile() generator, 137
- RedirectResponse class, 139, 140
- Redoc, 42–44
- Regular expression (RegEx), 59, 82

- Remote Procedure Call (RPC), 16

- Render Cloud

- dashboard, 281
 - FastAPI app, 279
 - features, 279
 - GitHub repository, 280
 - sign up, 280

- __repr__() method, 70

- Representational State Transfer (REST), 17, 91

- architecture, 13, 16, 28
 - drawback, 226
 - vs. WebSocket, 212

- response_model

- @app.post() decorator, 123
 - attribute, 135
 - body parameters, 123
 - Inventory_val, 123
 - JSON Schema, 122
 - operation decorator, 124
 - output data, 124
 - parameter, 95, 122
 - POST request, 122

- response_model_exclude, 124

- response_model_exclude_unset, 124

- response_model_include, 124

- Response object, 121

- Response status code

- client error responses, 132
 - informational responses, 132
 - redirection messages, 132
 - server error responses, 132, 133
 - status_code parameter, 133
 - successful responses, 132

INDEX

Response types

- FileResponse, 138
- HTMLResponse, 135
- json-encoder, 134
- JSONResponse, 134, 136
- media type, 134
- RedirectResponse, 139, 140
- StreamingResponse, 136, 137

REST constraints

- cacheability, 18
- client-server, 17
- code on demand, 18
- implementation
 - advantages, 18
- layered system, 18
- Roy Fielding, 17
- statelessness, 17
- uniform interface, 17

Route-based version, 31

Router package, 189–191

Roy Fielding, 17

run() function, 32, 33, 35

S

Schema, 231

Schema Definition Language
(SDL), 227, 228

schema_extra property, 76

schema_json() method, 71

SecretStr, 82

Security

- authentication, 248
- authorization, 248

basic access

authentication, 248–250

basic security dependency, 249

development process, 248

OAuth, 250–252

OAuth2PasswordBearer (*see*
OAuth2PasswordBearer)

select() method, 166, 168

Server-side script, 30

Session object, 158, 159

Sessionmaker() function, 158

set_cookie() method, 125, 126

/setcookie path, 126, 128

Simple Object Access Protocol
(SOAP), 16

Single file app, 182–184

sniffio package, 26

SQLAlchemy, 78, 251

@app.delete(), 163

@app.get(), 161

@app.post(), 160, 161

@app.put(), 162

async (*see* async in
SQLAlchemy)

connect to a database, 157

Core, 163

CRUD operations, 157

DB-API driver module, 157

declarative_base class, 78

dependency function, 159

engine object, 157

GET operation, 163

models, 78, 79

MySQL, 157

- object-relational mapper
 - API, 157
 - ORM model, 158
 - POST operation, 161
 - PUT operation, 162
 - pydantic model, 159, 160
 - session object, 158, 159
 - SQLite database, 157
- SQL data types, 156
- SQLite, 144, 146, 148–150, 152–154, 156, 157, 164, 170, 173, 179, 256, 268, 271
- SQLite Viewer extension, 146
- src tag, 108
- Starlette, 22, 23
- Starlette–Python’s ASGI toolkit, 21
- startup_event() function, 239
- Statelessness, 17
- /static route, 107
- Static assets, 107, 108
- StaticFiles class, 108, 191
- Static image, 111–113
- static-img.html, 112
- static-js.html, 109
- status_code parameter, 132
- Status code constants, 134
- Strawberry GraphQL
 - Book class, 232
 - browser-based user
 - interface, 234
 - FastAPI, 232
 - FastAPI route, 233
 - GraphQL object, 233
 - IDE, 234
 - Insomnia, GraphQL test, 236
 - mutation class, 236, 237
 - parameters, mutation
 - function, 236
 - PIP command, 232
 - Query Book type, 233
 - query, GraphQL IDE, 234
 - query result, 235
- StreamingResponse, 136, 137
- String-searching
 - algorithms, 59
- Student model, 80, 81
- Stylesheets, 107, 113
- Subscriptions, 230, 231, 233
- Suppliers model, 87
- Swagger documentation, 47–49, 57, 63, 64, 130, 133, 188
- Swagger interface, 77, 83
- Swagger tool’s interface, 81
- Swagger UI, 36, 66
 - Curl representation, 41
 - documentation works, 37
 - Execute button, 40
 - Form parameters, 119
 - index() function, 38
 - JSON response, 42
 - path operations, 37
 - path parameters, 37, 39
 - POST operation, 118, 149
 - query parameters, 199
 - Server response, 120
 - two APIs, single app, 184
 - user() function, 41
 - Uvicorn server, 37

T

- Table class methods, 166, 167
- Template engine, 96–97, 101, 107
- Template inheritance, 97
- TemplateResponse()
 - method, 99, 100
- TestClient object, 260, 261, 263, 264
- Testing
 - AsyncClient, 271
 - databases
 - live database, 265
 - override dependency, 265, 266
 - override get_db(), 266, 268, 269
 - setting up, 268
 - GET operation function, 260
 - HTTPX client library, 260
 - __init__.py file, 261
 - POST operation function, 260, 262
 - PyTest, 260
 - run, 262
 - test_list() function, 261, 262
 - test_main.py file, 261
 - WebSocket, 263, 264
- test_list() function, 261, 262
- Top-level domain (TLD), 82
- Traditional type, 93
- TypeError exception, 3
- Type hints, 4–6, 47
- Type parsing, 48, 49
- typing-extensions module, 26

U

- UJSONResponse, 136
- Unauthorized user, 15
- Uniform interface, 17, 19
- Uniform Resource Identifier (URI), 17, 19, 157, 212, 275
- update_book() function, 152, 156, 162, 169
- update() method, 167, 169
- url_for() function, 108, 111, 113
- URL validation, 85
- User-defined exception, 246
 - @app.exception_handler(), 246
 - 404 error code, 247
 - MyException class, 245
 - normal path operation, 247
 - status code 406, 247
- Uvicorn, 12, 21, 23, 24, 274, 276, 279
 - package, 13, 24
 - server, 101, 110, 218
- uvicorn.run() function, 13

V

- Validation, 82–85
 - @validator decorator, 85, 86
- Virtual machines (VMs), 282
- VS Code editor, 5

W

- watchfiles package, 26
- Web API, 16, 17, 25, 29, 226

- Web Server Gateway Interface (WSGI), 11
- `websocket_connect()` method, 264
- WebSockets
 - asyncio loop, 214
 - client, 214, 215
 - action, 221
 - JavaScript, 218
 - client-side form, 219
 - close WebSocket
 - connection, 220
 - connection, 213
 - event handler JavaScript
 - function, 219
 - FastAPI, 215, 216, 218, 220
 - handler coroutine, 224
 - handler function, 219
 - `hello()`, 213
 - Insomnia, 221, 222
 - multiclient chat
 - application, 222–226
 - protocol, 212
 - Python, 213
 - vs.* REST, 212
 - `serve()` coroutine, 213
 - server-side code, 214
 - support, 22
 - URL route, 216
 - test function, 263, 264
- Web template, 92, 96
- WSGIMiddleware, 239, 240

X

- X option, 66
- XML/JSON representation, 20
- “X-Web-Framework”, 129

Y, Z

- YAML, 286, 289
- yield statement, 136, 137