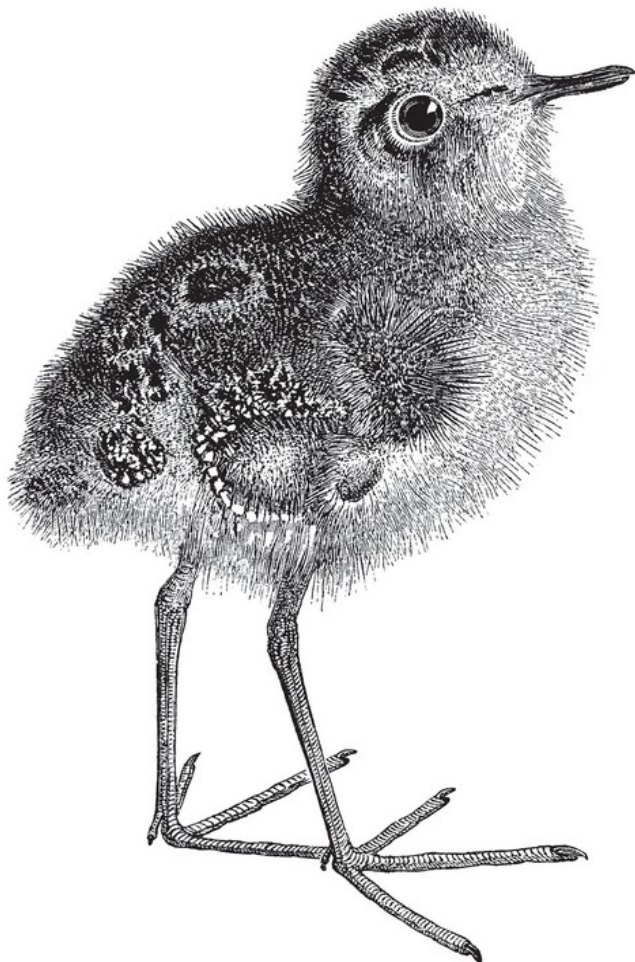


O'REILLY®

FastAPI

Modern Python Web Development



Early
Release

RAW &
UNEDITED

Bill Lubanovic

FastAPI

FIRST EDITION

Modern Python Web Development

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Bill Lubanovic

FastAPI

by Bill Lubanovic

Copyright © 2023 Bill Lubanovic. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales
promotional use. Online editions are also available for most titles
(<http://oreilly.com>). For more information, contact our corporate/institutional
sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Amanda Quinn
- Development Editor: Corbin Collins
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- November 2023: First Edition

Revision History for the Early Release

- 2022-09-14: First Release
- 2023-01-31: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098135508> for release
details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. FastAPI, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author(s) and do not represent the publisher's views. While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-13550-8

Dedication

To the loving memory of my parents Bill and Tillie, and my wife Mary. I miss you.

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the Preface of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

This is a pragmatic introduction to FastAPI — a modern Python web framework.

It’s also a story of how, now and then, the bright and shiny objects that we stumble across can turn out to be very useful. A silver bullet is nice to have when you encounter a werewolf. (And you will encounter werewolves later in this book.)

I started programming scientific applications in the mid 1970s. But not long after I first met UNIX and C on a PDP-11 in 1977, I had a feeling that this UNIX thing might catch on.

In the 80s and early 90s, the Internet was still non-commercial, but already a good source for free software and technical info. But when a “web browser” called Mosaic was distributed on the baby open Internet in 1993, I had a feeling that this Web thing might catch on.

When I started my own web development company a few years later, my tools were the usual suspects at the time — PHP, HTML, and Perl. On a

contract job a few years later, I finally experimented with Python, and was surprised how quickly I was able to access, manipulate, and display data. In some spare time over two weeks, I was able to replicate most of a C application that had taken four developers a year to write. Now I had a feeling that this Python thing might catch on.

After that, most of my work involved Python and its web frameworks, mostly Flask and Django. I particularly liked the simplicity of Flask, and preferred it for many jobs. But just a few years ago, I spied something glinting in the underbrush — a new Python web framework called FastAPI, written by Sebastián Ramirez.

As I read his (excellent) [documentation](#), I was impressed by the design and thought that had gone into it. In particular, his [history](#) page showed how much care he had spent evaluating alternatives. This was not an ego project or a fun experiment, but a serious framework for real-world development. Now I had a feeling that this FastAPI thing might catch on.

I wrote a biomedical API site with FastAPI, and it went so well that a team of us rewrote our old core API with FastAPI in the next year. This is still in production, and has held up well. Our group learned the basics that you'll read in this book, and all felt that we were writing better code, faster, with fewer bugs. And by the way, some of us had not written in Python before, and only I had used FastAPI.

So when I had an opportunity to suggest a followup to my *Introducing Python* book to O'Reilly, FastAPI was at the top of my list. In my opinion, FastAPI will have at least the impact that Flask and Django have had, and maybe more.

As I mentioned above, the FastAPI website itself provides world-class documentation, including many details on the usual web topics — databases, authentication, deployment, and so on. So why write a book?

This book isn't meant to be exhaustive because, well, that's exhausting. It is meant to be useful — to help you quickly pick up the main ideas of FastAPI and apply them. I will point out various techniques that required some sleuthing, and offer advice on day-to-day best practices.

I start each chapter with a **Preview** of what's coming. Next, I try not to forget what I just promised, with details and random asides. Finally, there's a brief digestible **Review**.

As the saying goes, “These are the opinions on which my facts are based.” Your experience will be unique, but I hope that you will find enough of value here to become a more productive web developer.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreilymedia/title_title.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*FastAPI* by Bill Lubanovic (O'Reilly). Copyright 2023 Bill Lubanovic, 978-1-098-13550-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

NOTE

For more than 40 years, [O'Reilly Media](#) has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://learning.oreilly.com/library/view/fastapi/9781098135492>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>

Follow us on Twitter: <https://twitter.com/oreillymedia>

Watch us on YouTube: <https://www.youtube.com/oreillymedia>

Part I. What's New?

The world has benefited greatly from the invention of the World Wide Web by Sir Tim Berners-Lee¹, and the Python programming language by Guido van Rossum.

The only tiny problem is that a nameless computer book publisher often puts spiders and snakes on its relevant Web and Python covers. If only the Web had been named the World Wide *Woof* (cross-threads in weaving, also called *weft*), and Python were *Pooch*, this book might have had a cover like this:



Figure I-1. *FastAPI: Modern Pooch Woof Development*

But I digress².

This book is about:

- *The Web*: An especially productive technology, how it has changed, and how to develop software for it now
- *Python*: An especially productive web development language
- *FastAPI*: An especially productive Python web framework

The two chapters in this first part discuss emerging topics in the Web and Python: services and APIs, concurrency, layered architectures, and big big data.

Part II is a high-level tour of FastAPI, a fresh Python web framework that has good answers to the questions posed in Part I.

Part III rummages much deeper through the FastAPI toolbox, including tips learned during production development.

Finally, Part IV provides a gallery of FastAPI web examples. They use a common data source — imaginary creatures — that may be a little more interesting and cohesive than the usual random expositions. These should give you a starting point for particular applications.

¹I actually shook his hand once. I didn't wash mine for a month, but I'll bet he did right away.

²Not for the last time.

Chapter 1. The Modern Web

The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past.

Tim Berners-Lee

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

Once upon a time, the web was small and simple. Developers had such fun throwing PHP, HTML, and MySQL calls into single files and proudly telling everyone to check out their website. But the web grew over time to zillions, nay, squillions of pages — and the early playground became a metaverse of theme parks.

In this chapter, I’ll point out some areas that have become ever more relevant to the modern web:

- Services and APIs

- Concurrency
- Layers
- Data

The next chapter will show what Python offers in these areas. After that, we'll dive into the FastAPI web framework and see what it has to offer.

Services and APIs

The web is a great connecting fabric. Although there is still much activity on the *content* side — HTML, JavaScript, images, and so on — there's an increasing emphasis on the *APIs* (Application Programming Interfaces) that connect things.

Commonly, a *web service* handles low-level database access and middle-level business logic (often lumped together as a *backend*), while JavaScript or mobile apps provide a rich top-level *frontend* (interactive user interface). These fore and aft worlds have become more complex and divergent, usually requiring developers to specialize in one or the other. It's harder to be a *full stack* developer than it used to be^{[1](#)}.

These two worlds talk to each other using APIs. In the modern web, API design is as important as the design of web sites themselves. An API is a contract, similar to a database schema. Defining and modifying APIs is now a major job.

Kinds of APIs

Each API defines some:

- *Protocol*: Control structure
- *Format*: Content structure

Different API methods have developed as technology has evolved from isolated machines, to multitasking systems, to networked servers. You'll probably run across one or more of these at some point, so the following is a

brief summary before getting to *HTTP* and its friends, which are featured in this book:

- Before networking, an API usually meant a very close connection, like a function call to a *library* in the same language as your application — say, calculating a square root in a math library.
- *RPCs* (Remote Procedure Calls) were invented to call functions in other processes, on the same machine or others, as though they were in the calling application. A popular current example is [gRPC](#).
- *Messaging* sends small chunks of data in pipelines among processes. Messages may be verb-like commands, or may just indicate noun-like *events* of interest. Current popular messaging solutions, which vary broadly from toolkits to full servers, include [Kafka](#), [RabbitMQ](#), [NATS](#), and [ZeroMQ](#). Communication can follow different patterns:
 - Request-response: One:one, like a web browser calling a web server.
 - Publish-subscribe, or *pub-sub*: A *publisher* emits messages, and *subscribers* act on each according to some data in the message, like a subject.
 - Queues: Like pub-sub, but only one of a pool of subscribers grabs the message and acts on it.

Any of these may be used alongside a web service — for example, performing some slow backend task like sending an email or creating a thumbnail image.

HTTP

Berners-Lee proposed three components for his World Wide Web:

- HTML: A language for displaying data

- HTTP: A client-server protocol
- URLs: An addressing scheme for web resources

Although these seem obvious in retrospect, they turned out to be a ridiculously useful combination. As the web evolved, people experimented, and some ideas, like the IMG tag, survived the Darwinian struggle. And as needs became more clear, people got serious about defining standards.

REST(ful)

One chapter in the Ph.D. [thesis](#) by Roy Fielding defined *REST* (**Representational State Transfer**) — an *architectural style*² for HTTP use. Although often referenced, it's been largely [misunderstood](#).

A roughly shared adaptation has evolved, and dominates the modern web. It's called *RESTful*, with these characteristics:

- Uses HTTP and client-server
- Stateless: Each connection is independent
- Cacheable
- Resource-based

A *resource* is data that you can distinguish and perform operations on. A web service provides an *endpoint* — a distinct URL and HTTP *verb* (action) — for each feature that it wants to expose. An endpoint is also called a *route*, because it routes the URL to a function.

Database users are familiar with the *CRUD* acronym of procedures — create, read, update, delete. The HTTP verbs are pretty CRUDdy:

- POST: Create (write)
- PUT: Modify completely (replace)
- PATCH: Modify partially (update)
- GET: Um, get (read, retrieve)

- DELETE: Uh, delete

A client sends a *request* to a RESTful endpoint with data in some area of an HTTP message:

- Headers
- The URL string
- Query parameters
- Body values

In turn, an HTTP *response* returns:

- An integer *status code* indicating:
 - 100s: Info, keep going
 - 200s: Success
 - 300s: Redirection
 - 400s: Client error
 - 500s: Server error
- Various headers
- A body, which may be empty, single, or *chunked* (in successive pieces)



At least one status code is an Easter egg: 418 (I'm a teapot) is supposed to be returned by a web-connected teapot, if asked to brew coffee.

You'll find many web sites and books on RESTful API design, all with useful rules of thumb. This book will dole some out on the way.

JSON and API Data Formats

Frontend applications can exchange plain ASCII text with backend web services, but how can you express data structures like lists of things?

Just about when we really started to need it, along came *JSON* (JavaScript Object Notation) — another simple idea that solves an important problem, and seems obvious with hindsight. Although the J stands for JavaScript, the syntax looks a lot like Python, too.

JSON has largely replaced older attempts like XML and SOAP. In the rest of this book, you'll see that JSON is the default web service input and output format.

JSON:API

The combination of RESTful design and JSON data formats is very common now. But there's still some wiggle room for ambiguity and nerd tussles. The recent [JSON:API](#) proposal aims to tighten specs a bit. This book will use the loose RESTful approach, but JSON:API or something similarly rigorous may be useful if you have significant tussles.

GraphQL

RESTful interfaces can be cumbersome for some purposes. Facebook designed [GraphQL](#) (Graph Query Language) to specify more flexible service queries. I won't go into GraphQL in this book, but you may want to look into it if you find RESTful design inadequate for your application.

Concurrency

Besides the growth of service orientation, the rapid expansion of the number of connections to web services requires ever better efficiency and scale.

We want to reduce:

- *Latency*: The upfront wait time.

- *Throughput*: The number of bytes per second between the service and its callers.

In the old web days³, people dreamed of supporting hundreds of simultaneous connections, then fretted about the “10K problem”, and now assume millions at a time.

The term *concurrency* doesn’t mean full parallelism. Multiple processing isn’t occurring in the same nanosecond, in a single CPU. Instead, concurrency mostly avoids *busy waiting*. CPUs are very zippy, but networks and disks are thousand to millions of times slower. So, whenever we talk to a network or disk, we don’t want to just sit there with a blank stare.

Normal Python execution is *synchronous*: one thing at a time, in the order specified by the code. Sometimes we want to be *asynchronous*: do a little of one thing, then a little of another thing, back to the first thing, and so on. If all our code uses the CPU to calculate things (*CPU-bound*), there’s really no spare time to be asynchronous. But if we perform something that makes the CPU wait for some external thing to complete (*I/O-bound*), then we can be asynchronous. Asynchronous systems provide an *event loop*: requests for slow operations are sent and noted, but we don’t hold up the CPU waiting for their responses. Instead, some immediate processing is done on each pass through the loop, and any responses that came in during that time are handled in the next pass.

The effects can be dramatic. CPU and memory access take nanoseconds, but network or disk access are thousands to millions of times slower. Later in this book, you’ll see how FastAPI’s support of asynchronous processing makes it much faster than typical web frameworks.

It isn’t magic. You still have to be careful to avoid doing too much CPU-intensive work during the event loop, because that will slow down everything.

Later in this book, you’ll see the uses of Python’s `async` and `await` keywords, and how FastAPI lets you mix both synchronous and asynchronous processing.

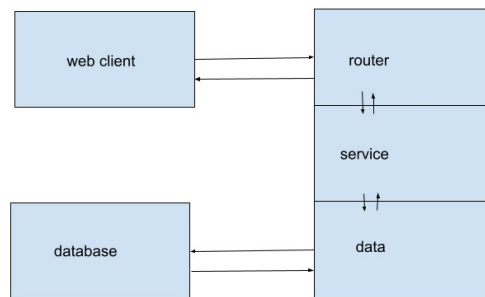
Layers

Shrek fans may remember he noted his layers of personality, to which Donkey replied, “Like an onion?”



Well, if ogres and tearful vegetables can have layers, then so can software. To manage size and complexity, many applications have long used a so-called *three-tier* model⁴. This actually isn’t terribly new. Terms differ⁵, but for this book I’m using the following simple separation and terms:

- *Router*: Get client requests, call service, return responses
- *Service*: The business logic, which calls the data layer when needed
- *Data*: Access to data stores and other services



These will help you to scale your site without having to start from scratch. They’re not laws of quantum mechanics, so consider them guidelines for this book’s exposition.

The layers talk to one another via APIs. These can be simple function calls to separate Python modules, but could access external code via any method. As I showed earlier, this could include RPCs, messages, and so on. In this book, I'm assuming a single web server, with Python code importing other Python modules. The separation and information hiding is handled by the modules.

The Router layer is the one that users see, via *client* applications and APIs. We're usually talking about a RESTful web interface, with URLs, and JSON-encoded requests and responses. But there may also be text (or *CLI*, command line interface) clients. Python Router code may import Service-layer modules, but should not import Data modules.

The Service layer contains the actual details of whatever this web site provides. This essentially looks like a *library*. It imports Data modules to access databases and external services, but should not know the details.

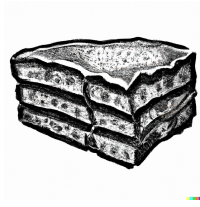
The Data layer provides the service layer access to data, through files or client calls to other services. There may also be alternative Data layers, communicating with a single Service layer.

Why do this? Among many reasons, each layer can be:

- Written by specialists.
- Tested in isolation.
- Replaced or supplemented: You might add a second Router layer, using a different API such as gRPC, alongside a web one.

Follow one rule from *Ghostbusters*: *don't cross the streams*. That is, don't let web details leak out of the Routing layer, or database details out of the Data layer.

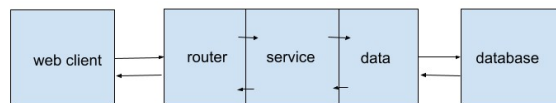
You can visualize “layers” as a vertical stack, like a cake in the British Baking Show^{[6](#)}.



Reasons for separation:

- If you don't, expect a hallowed web meme: *Now you have two problems.*
- Once they're mixed, later separation will be *very* difficult.
- You'll need to know two or more specialties to understand and write tests if code logic gets muddled.

By the way, even though I call them *layers*, you don't need to assume that one layer is “above” or “below” another, and that commands flow with gravity. Vertical chauvinism! You could also view layers as sideways-communicating boxes:



However you visualize them, the *only* communication paths between the boxes/layers are the arrows (APIs). This is important for testing and debugging. If there are undocumented doors into a factory, the night watchman will inevitably be surprised.

The arrows between the web client and router layer use HTTP or HTTPS to transport mostly JSON text. The arrows between the data layer and database use a database-specific protocol and carry SQL (or other) text. The arrows between the layers themselves are function calls carrying data models.

Also, the recommended data formats flowing through the arrows are:

- Client \Leftrightarrow Router: RESTful HTTP with JSON
- Router \Leftrightarrow Service: Models
- Service \Leftrightarrow Data: Models
- Data \Leftrightarrow Databases and services: Specific APIs

Based on my own experience, this is how I've chosen to structure the topics in this book. It's workable, and has scaled to fairly complex sites, but isn't sacred. You may have a better design! However you do it, the important points are:

- Separate domain-specific details.
- Define standard APIs between the layers.
- Don't cheat, don't leak.

Sometime's it's a challenge deciding which layer is the best home for some code. For example, chapter 10 looks at “auth” requirements, and how to implement them — as an extra layer between Router and Service, or within one of them. Software development is sometimes as much art as science.

Data

The web has often been used as a frontend to relational databases, although many other ways of storing and accessing data have evolved, such as NoSQL or NewSQL databases.

But beyond databases, *ML* (machine learning, or *deep learning*, or just *AI*) is fundamentally remaking the technology landscape. The development of

large models requires *lots* of messing with data — traditionally called *ETL* (extraction / transformation / loading).

As a general purpose service architecture, the web can help with many of the fiddly bits of ML systems.

Review

The web uses many APIs, but especially RESTful ones. Asynchronous calls allow better concurrency, which makes things faster. Web service applications are often large enough to divide into layers. Data has become a major area in its own right. All of these concepts are addressed in the Python programming language, coming in the next chapter.

[1](#) I gave up trying a few years ago.

[2](#) “Style” means a higher level pattern, like “client-server”, rather than a specific design.

[3](#) Around when caveman played hacky sack with giant ground sloths.

[4](#) Choose your own dialect: tier/layer, tomato/tomahto/arrigato.

[5](#) You’ll often see the term *MVC* (Model View Controller) and variations. Often accompanied by religious wars, toward which I’m agnostic.

[6](#) As viewers know, if your layers get sloppy, you may not return to the tent the next week.

Chapter 2. Modern Python

It's all in a day's work for "Confuse-a-Cat".

Monty Python

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

Python evolves to keep up with our changing technical world. This chapter discusses specific Python features that apply to issues in the previous chapter, and a few extras:

- Tools
- APIs and services
- Type hinting and variables
- Concurrency
- Data Structures

- Web frameworks

Tools

Every computing language has:

- The core language, and built-in “standard” packages
- Ways to add external packages
- Recommended external packages
- An environment of development tools

The following sections list what’s required or recommended for this book.

These may change over time! Python packaging and development tools are moving targets, and better solutions come along now and then.

Getting Started

You should be able to write and run a Python program like this:

Example 2-1. this.py

```
def paid_promotion():  
    print("(that calls this function!)")  
  
print("This is the program")  
paid_promotion()  
print("that goes like this.")
```

To execute this program from the command line in a text window or terminal, I’ll use the convention of a \$ *prompt* (your system begging you to type something, already). What you type after the prompt is shown in **bold print**. If you saved the little program above to a file named *this.py*, then you can run it like this:

```
$ python this.py
This is the program
(that calls this function!)
that goes like this.
```

Some code examples use the interactive Python interpreter, which is what you get if you just type python:

```
$ python
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first few lines are specific to your operating system and Python version. The >>> is your prompt here. A handy extra feature of the interactive interpreter is that it will print the value of a variable for you if you type its name:

```
>>> wrong_answer = 43
>>> wrong_answer
43
```

This also works for expressions:

```
>>> wrong_answer = 43
>>> wrong_answer - 3
40
```

If you're fairly new to Python, or would like a quick review, read the next few sections.

Python Itself

You will need, as a bare minimum, Python 3.7. This includes features like type hints and asyncio, which are core requirements for FastAPI. I recommend using at least Python 3.9, which will have a longer support lifetime. The standard source for Python is python.org.

Package Management

You will want to download external Python packages, and install them safely on your computer. The classic tool for this is [pip](https://pip.pypa.io).

But how do you download this downloader? If you installed Python from python.org, you should already have pip. If not, follow the instructions at the pip site to get it. Throughout this book, as I introduce a new Python package, I'll include the pip command to download it.

Although you can do a lot with plain old pip, it's likely that you'll also want to use virtual environments, and consider an alternative tool like poetry.

Virtual Environments

Pip will download and install packages, but where should it put them? Although standard Python and its included libraries are usually installed in some “standard” place on your operating system, you may not (and probably should not) be able to change anything there. Pip uses a default directory other than the system one, so you won't step on your system's standard Python files. You can change this; see the pip site for details for your operating system.

But it's common to work with multiple versions of Python, or make installations specific to a project, so you know exactly which packages are in there. To do this, Python supports *virtual environments*. These are just directories (“folders” in the non-Unix world) into which pip writes downloaded packages. When you *activate* a virtual environment, your shell (main system command interpreter) looks there first when loading Python modules.

The program for this is [venv](https://venv.pypa.io), and it's been included with standard Python since version 3.4.

Let's make a virtual environment called `venv1`. You can run the `venv` module as a standalone program:

```
$ venv venv1
```

or as a Python module:

```
$ python -m venv venv1
```

To make this your current Python environment, run this shell command (on Linux or Mac; see the `venv` docs for Windows and others):

```
$ source venv1/bin/activate
```

Now, anytime you run `pip install`, it will install packages under `venv1`. And when you run Python programs, that's where your Python interpreter and modules will be found.

To *de*-activate your virtual environment, type a control-d (Linux or Mac), or `deactivate` (Windows).

You can create alternative environments like `venv2`, and deactivate/activate to step between them. (although I hope you have more naming imagination than me).

Poetry

This combination of `pip` and `venv` is so common that people started combining them to save steps, and avoid that `source` shell wizardry. One such package is [pipenv](#), but a newer rival called [poetry](#) is becoming more popular.

Having used `pip`, `pipenv`, and `poetry`, I now prefer `poetry`. Get it with `pip install poetry`. Poetry has many subcommands, such as `poetry add` to

add a package to your virtual environment, `poetry install` to actually download and install it, and so on. Check the poetry site or run the poetry command for help.

Besides downloading single packages, pip and poetry manage multiple packages in configuration files: `requirements.txt` for pip, and `pyproject.toml` for poetry. They don't just download packages, but also manage the tricky dependencies that packages may have on other packages. You can specify desired package versions as minima, maxima, ranges, or exact values (also known as *pinning*). This can be important as your project grows and the packages that it depends on change. You may need a minimum version of a package if a feature that you use first appeared there, or a maximum if some feature was dropped.

Source Formatting

This is less important, but helpful. Avoid code formatting (“bikeshedding”) arguments with a tool that massages source into a standard, non-weird format. One good choice is [black](#). Install it with `pip install black`.

Testing

Testing will be covered in detail in Chapter 13. Although the standard Python test package is `unittest`, the industrial-strength Python test package used by most Python developers is [pytest](#). Install it with `pip install pytest`.

Source Control and Continuous Integration (CI)

The almost-universal solution for source control now is *git*, with storage repositories (“repos”) at sites like github and gitlab. This isn't specific to Python or FastAPI, but you'll very likely spend a lot of your development time with git. The [pre-commit](#) tool runs various tests on your local machine such as black and pytest) before committing to git. After pushing to a remote git repo, more CI tests may be run there.

Chapters 13 (Testing) and 16 (Troubleshooting) will have more details.

Web Tools

Chapter 3 will show how to install and use the main Python web tools used in this book:

- FastAPI: The web framework itself
- Uvicorn: An asynchronous web server
- Httpie: A text web client, similar to curl
- Requests: A synchronous web client package
- Httpx: A synchronous/asynchronous web client package

APIs and Services

Python’s modules and packages are essential for creating large applications that don’t become [“big balls of mud”](#). Even in a single-process web service, we can maintain the separation discussed in Chapter 1 by the careful design of modules and imports.

Python’s built-in data structures are extremely flexible, and very tempting to use everywhere. But in the coming chapters you’ll see that we can define higher-level *models* to make our inter-layer communication cleaner. These models rely on a fairly recent Python addition called *type hinting*. Let’s get into that, but first with a brief aside into how Python handles *variables*. This won’t hurt.

Variables Are Names

The term *object* has many definitions in the software world — maybe too many. In Python, an object is a data structure that wraps every distinct piece of data in the program, from an integer like 5, to a function, to anything that you might define. It specifies, among other bookkeeping info:

- A unique *identity* value
- The low-level *type* that matches the hardware

- The specific *value* (physical bits)
- A *reference count* of how many variables refer to it

Python is *strongly typed* at the object level (its *type* above doesn't change, although its *value* might). An object is termed *mutable* if its value may be changed, *immutable* if not.

But at the *variable* level, Python differs from many other computing languages, and this can be confusing.

In many other languages, a *variable* is essentially a direct pointer to an area of memory that contains a raw *value*, stored in bits that follow the computer's hardware design. If you assign a new value to that variable, the language overwrites the previous value in memory with the new one.

That's direct and fast. The compiler keeps track of what goes where. It's one reason why languages like C are faster than Python. As a developer, you need to ensure that you only assign values of the correct type to each variable.

Now, here's the big difference with Python: a Python variable is just a *name* that is temporarily associated with a higher-level *object* in memory.

If you assign a new value to a variable that refers to an immutable object, you're actually creating a new object that contains that value, and then getting the name to refer to that new object. The old object (that the name used to refer to) is then free, and its memory can be reclaimed if no other names are still referring to it (i.e., its reference count is zero).

In *Introducing Python* (O'Reilly, 2020), I compare objects to plastic boxes sitting on memory shelves, and names/variables to sticky notes on these boxes. Or you can picture names as tags attached by strings to those boxes.

Usually, when you use a name, you assign it to one object and it stays attached. Such simple consistency helps you to understand your code. A variable's *scope* is the area of code in which a name refers to the same object — such as within a function. You can use the same name in different scopes, but each one refers to a different object.

Although you can make a variable refer to different things throughout a Python program, that isn't necessarily a good practice. Without looking, you don't know if name *x* on line 100 is in the same scope as name *x* on line 20.

(By the way, `x` is a terrible name. We should pick names that actually confer some meaning.)

Type Hints

All of this background has a point.

Python 3.6 added *type hints* to declare the type of object to which a variable refers. These are *not* enforced by the Python interpreter as it's running! Instead, they can be used by various tools to ensure that your use of a variable is consistent. The standard type checker is called *mypy*, and I'll show you how it's used later.

A type hint may just seem like a nice thing, like many “lint” tools used by programmers to avoid some mistakes. For instance, it may remind you that your variable `count` refers to a Python object of type `int`. But hints, although they're optional and unenforced notes (literally, “hints”), turn out to have unexpected uses. Later in this book, you'll see how FastAPI adapted the Pydantic package to make very clever use of type hinting.

The addition of type declarations may be a trend in other, formerly typeless, languages. For example, many JavaScript developers have moved to [TypeScript](#).

Data Structures

You'll get details in Chapter 5.

Web Frameworks

Among other things, a web framework translates between HTTP bytes and Python data structures. It can save you a lot of effort. On the other hand, if some part of it doesn't work as you need it to, you may need to hack a solution around it.

[WSGI](#) (the Web Server Gateway Interface) is a synchronous Python [standard specification](#) to connect application code to web servers.

Concurrency has become more important in recent years. As a result, the Python [ASGI](#) (Asynchronous Standard Gateway Interface) specification was developed.

Django

[Django](#) is a full-featured web framework, that tags itself as *the web framework for perfectionists with deadlines*. It was announced by Adrian Holovaty and Simon Willison in 2003, and named after Django Reinhardt, a twentieth century Belgian jazz guitarist. Django is often used for database-backed corporate sites. I'll include more details on Django in Chapter 7.

Flask

In contrast, [Flask](#), introduced by Armin Ronacher in 2010, is a *micro framework*. Chapter 7 will have more information on Flask, and how it compares with Django and FastAPI.

FastAPI

Finally we meet FastAPI, the subject of this book. Although FastAPI was published by Sebastián Ramírez in 2018, it has already climbed to third place of Python web frameworks, behind Flask and Django. This [chart](#) of github stars (as of July 8, 2022) shows that it may pass them at some point:



tiangolo/fastapi django/django pallets/flask



update

tiangolo/fastapi

Total stars

☆ 46,973

Avg. stars/day

35.91

Max. stars/day

390

django/django

Total stars

☆ 64,994

Avg. stars/day

17.45

Max. stars/day

3,847

pallets/flask

Total stars

☆ 59,800

Avg. stars/day

13.36

Max. stars/day

1,580

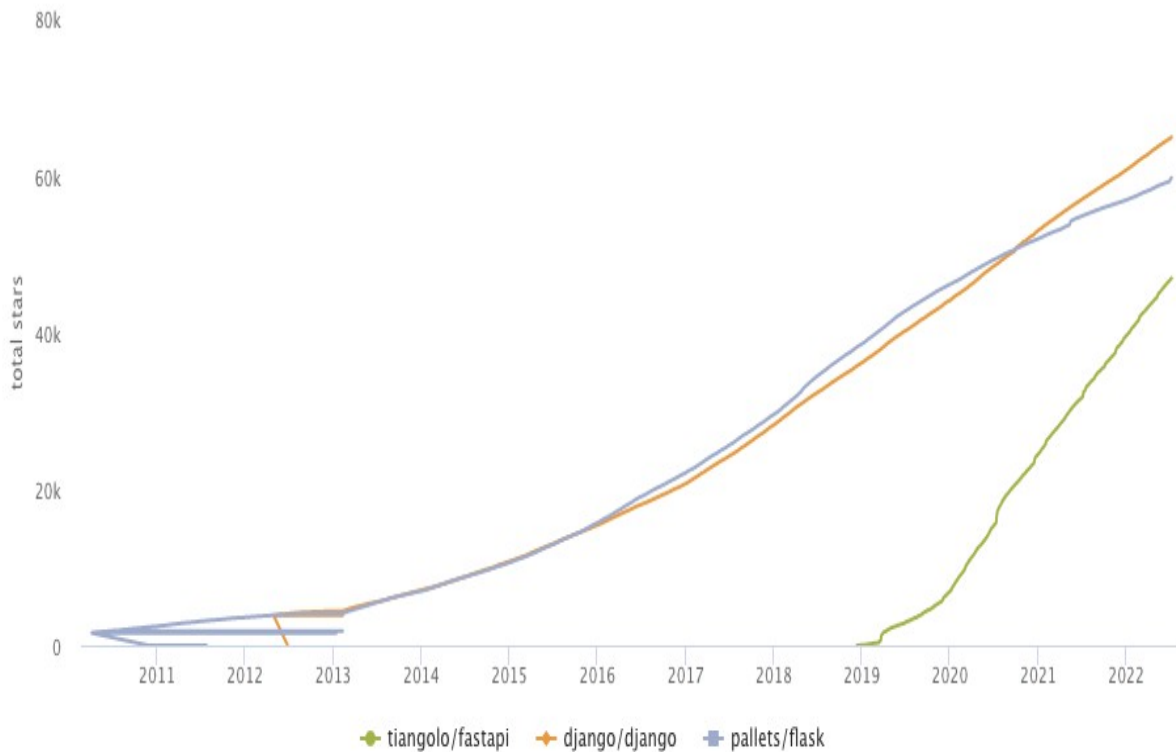


Figure 2-1. Github Stars

After careful investigation into [alternatives](#), Sebastián came up with a [design](#) that was heavily based on two third-party Python packages:

- *Starlette* for web stuff
- *Pydantic* for data stuff

And he added his own ingredients and special sauces to the final product. You'll see what I mean in the next chapter.

Review

This chapter covered a lot of ground related to today's Python:

- Useful tools for a Python web developer
- The prominence of APIs and Services
- Python's type hinting, objects, and variables
- Concurrency
- Data structures for web services
- Web frameworks

Part II. A FastAPI Tour

The chapters in this part provide a whatever-thousand foot/meter view of FastAPI — more like a drone than a spy satellite. They cover the basics quickly, but stay above the water line to avoid drowning you in details. The chapters are relatively short, and are meant to provide context for the depths of Part 3.

After you get used to the ideas in this part, Part III zooms down into those details. That's where you can do some serious good, or damage. No judgement, it's up to you.

Chapter 3. Overview

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

Sebastián Ramírez

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

FastAPI was announced in 2018 by [Sebastián Ramírez](#). It’s more “modern” in many senses than most Python web frameworks — taking advantage of some features that have been added to Python 3 in the last few years. This chapter is a quick look at FastAPI’s main features.

What Is FastAPI?

Like any web framework, FastAPI helps you to build web applications. Every framework is designed to make some operations easier — by features, omissions, and defaults. As the name implies, FastAPI targets development

of web APIs, although you can use it for traditional web content applications as well.

Some advantages claimed by the web site include:

- *Performance*: As fast as node and go lang in some cases, unusual for Python frameworks
- *Faster development*: No sharp edges or oddities
- *Better code quality*: Type hinting and models help reduce bugs
- *Autogenerated documentation and test pages*: Much easier than hand editing OpenAPI descriptions

FastAPI uses:

- Python type hints
- Starlette for the web machinery, including async support
- Pydantic for data definitions
- Special integration to leverage and extend the others

Together, it's a pleasing development environment for web applications, especially RESTful web services.

A FastAPI Application

Let's write a teeny FastAPI application — a web service with a single endpoint. For now, we're just in what I've called the "Router" layer, only handling web requests and responses. First, install the basic Python packages that we'll be using:

- The [FastAPI](#) framework: `pip install fastapi`
- The [Uvicorn](#) web server: `pip install uvicorn`

- The [Httpie](#) text web client: `pip install httpie`
- The [Requests](#) synchronous web client package: `pip install requests`
- The [Httpx](#) synchronous/asynchronous web client package: `pip install httpx`

Although [Curl](#) is the best known text web client, I think Httpie is easier to use. Also, it defaults to JSON encoding and decoding, which is a better match for FastAPI. Later in this chapter, you'll see a screenshot that includes the syntax of the Curl command line needed to access a particular endpoint.

We'll shadow an introverted web developer and save this code as file *hello.py*:

Example 3-1. A Shy Endpoint

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Hello? World?"
```

There are two main parts to notice:

- `@app.get("/hi")` is the *path decorator*. It tells FastAPI that:
 - A request for the URL `"/hi"` on this server should be directed to the following function.
 - This applies only to the HTTP GET verb. You can also respond to a `"/hi"` URL sent with the other HTTP verbs (PUT, POST, etc.), each with a separate function.
- `def greet()` is the *path function*. It's your main point of contact with HTTP requests and responses. In this example, it

has no arguments, but the following sections show that there's much more under the FastAPI hood.

For now, let's just see how we can test this really basic example.

FastAPI itself does not include a web server, but recommends Uvicorn. Start the web server:

```
$ uvicorn hello:app --reload
```

That `--reload` tells it to restart the web server if *hello.py* changes, and that's what we're going to do.

The `hello` above refers to the *hello.py* file, and the `app` is the FastAPI variable in it. That's all that Uvicorn needs to get the right file and load the right stuff.

This will use port 8000 on your machine (named `localhost`) by default. Now the server has a single endpoint and is ready for requests.

Let's try all three clients.

- For the browser, type the URL in the top location bar.
- For Httpie, type the command shown (the `$` stands for whatever command prompt you have for your system shell).
- For Requests, use Python in interactive mode, and type after the `>>>` prompt.

As mentioned in the preface, what you type is in a

```
bold monospaced font
```

and the output is in a

```
normal monospaced font
```

Browser:

```
http://localhost:8000/hi
```

Requests:

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi")
>>> r.json()
'Hello? World?'
```

Httpx is almost identical to Requests:

```
>>> import httpx
>>> r = httpx.get("http://localhost:8000/hi")
>>> r.json()
'Hello? World?'
```

NOTE

It doesn't matter if you use Requests or Httpx to test FastAPI routes. But chapter 13 shows cases where Httpx is useful when making other asynchronous calls. So the rest of the examples in this chapter use Requests.

Httpie:

```
$ http localhost:8000/hi
HTTP/1.1 200 OK
content-length: 15
content-type: application/json
```

```
date: Thu, 30 Jun 2022 07:38:27 GMT
server: uvicorn
```

```
"Hello? World?"
```

In Httpie, to omit the response headers and get only the response body, use the `-b` argument:

```
$ http -b localhost:8000/hi
"Hello? World?"
```

Also in Httpie, to get the full request as well as response, use `-v`:

```
$ http -v localhost:8000/hi
GET /hi HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1

HTTP/1.1 200 OK
content-length: 15
content-type: application/json
date: Thu, 30 Jun 2022 08:05:06 GMT
server: uvicorn

"Hello? World?"
```

In most examples in this book, I'll use the default Httpie output (response headers and body), so you can see everything that you're getting back.

HTTP Requests

That example included only one specific request: a GET request for the `/hi` URL on the server `localhost`, port `8000`.

There are two main parts to notice:

- `@app.get("/hi")` is the *path decorator*. It tells FastAPI that:
 - A request for the URL `/hi` on this server should be directed to the following function.
 - This applies only to the HTTP GET verb. You can also respond to a `/hi` URL sent with the other HTTP verbs (PUT, POST, etc.), each with a separate function.
- `def greet()` is the *path function*. Because it's "wrapped" by the preceding decorator, FastAPI can do many useful things here:
 - Provide arguments that come from any part of the HTTP request.
 - Provide arguments from user-defined functions.

Web requests squirrel different data in different parts of an HTTP request. From the sample request above, here's the HTTP request that the `http` command sent to the web server:

```
GET /hi HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1
```


The first line contains the path (`/hi`) and any query parameters (in this case, none), and the rest are HTTP headers. No body content was uploaded.

FastAPI unsquirrels these into handy definitions:

- Header: The HTTP headers
- Path: The URL
- Query: The query parameters (after the `?` at the end of the URL)
- Body: The HTTP body

NOTE

The way that FastAPI provides data from various parts of the HTTP requests is one of its best features, and an improvement on how most Python web frameworks do it. All the arguments that you need can be declared and provided directly inside the path function, using the definitions above (Path, Query, etc.), and by functions that you write. This uses a technique called *dependency injection*, which I'll illustrate as we go along, and expand on in chapter 6.

Let's make our earlier application a little more personal by adding a parameter called `who` that addresses that plaintive `Hello?` to someone.

We'll try different ways to pass this new parameter:

- In the URL *path*.
- As a *query* parameter, after the `?` in the URL.
- In the HTTP *body*.
- As an HTTP *header*.

URL Path

Edit *hello.py*:

Example 3-2. Return the Greeting Path

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/hi/{who}")
```

```
def greet(who):
```

```
    return f"Hello? {who}?"
```

Adding that `{who}` in the URL (after the `@app.get`) tells FastAPI to expect a variable named `who` at that position in the URL. FastAPI then assigns it to the `who` argument in the following `greet()` function. This shows coordination between the path decorator and the path function.

NOTE

Do not use an f-string for the amended URL string `"/hi/{who}"` here. The curly brackets are used by FastAPI itself to match URL pieces as path parameters.

Saving your updated *hello.py* file should cause uvicorn to read the updated file. (Otherwise, we'd create *hello2.py*, etc. and rerun Uvicorn each time.) Now test it with the various methods we've discussed earlier:

Browser:

```
localhost:8000/hi/Mom
```

Httpie:

```
$ http localhost:8000/hi/Mom
HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Thu, 30 Jun 2022 08:09:02 GMT
server: uvicorn

{"Hello? Mom?"}
```

Requests:

```
>>> import requests
>>> r = requests.get("localhost:8000/hi/Mom")
>>> r.json()
'Hello? Mom?'
```

In each case, the string "Mom" was passed as part of the URL, def to the path function, and returned as part of the response.

Query Parameters

Query parameters are the strings after the ? in a URL, separated by & characters. Edit *hello.py* again:

Example 3-3. Return the Greeting Query Parameter

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet(who):
    return f"Hello? {who}?"
```

The endpoint function is defined as `greet(who)` again, but `{who}` isn't in the URL on the previous line this time, so FastAPI now assumes that `who` is a query parameter.

Example 3-4. Test With Your Browser

```
localhost:8000/hi?who=Mom
```

Example 3-5. Test With HTTPie

```
$ http localhost:8000/hi?who=Mom
HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Thu, 30 Jun 2022 08:10:27 GMT
server: uvicorn
```

```
"Hello? Mom?"
```

You could have also called httpie with a query parameter argument (note the ==):

Example 3-6. Test with HTTPie and Params

```
$ http localhost:8000/hi who==Mom
```

You can have more than one of these arguments for Httpie, and that's easier to type if there are a lot of them.

Example 3-7. Test With Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi?who=Mom")
>>> r.json()
'Hello? Mom?'
```

There's another way to pass query parameters with requests, too:

Example 3-8. Test With Requests and Params

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi", params={"who": "Mom"})
>>> r.json()
'Hello? Mom?'
```

In each case, we provided the "Mom" string in a new way, and got it to the path function and the eventual response.

Body

We can provide path or query parameters to a GET endpoint, but not values from the request body. In HTTP, GET is supposed to be *idempotent*¹ — a computery term for *ask the same question, get the same answer*. HTTP GET is only supposed to return stuff. The request body is used to send stuff to the server when creating (POST) or updating (PUT or PATCH). Chapter 9 shows a way around this.

So, let's change the endpoint from a GET to a POST for this example².

Example 3-9. Return the Greeting Body

```
from fastapi import FastAPI, Body

app = FastAPI()

@app.post("/hi")
def greet(who:str = Body(embed=True)):
    return f"Hello? {who}?"
```

NOTE

That `Body(embed=True)` stuff is needed to tell FastAPI that, this time, we get the value of `who` from the JSON-formatted request body. The `embed` part means that it should look like `{ "who": "Mom" }` rather than just `"Mom"`.

Try it with `Httpie`, using `-v` to show the generated request body (and note the `single =` parameter to indicate JSON body data):

Example 3-10. Test With HTTPie

```
$ http -v localhost:8000/hi who=Mom
POST /hi HTTP/1.1
Accept: application/json, /;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 14
```

```
Content-Type: application/json
```

```
Host: localhost:8000
```

```
User-Agent: HTTPie/3.2.1
```

```
{  
  "who": "Mom"  
}
```

```
HTTP/1.1 200 OK
```

```
content-length: 13
```

```
content-type: application/json
```

```
date: Thu, 30 Jun 2022 08:37:00 GMT
```

```
server: uvicorn
```

```
"Hello? Mom?"
```

And finally, Requests, which uses its `json` argument to pass JSON-encoded data in the request body:

Example 3-11. Test With Requests

```
>>> import requests  
>>> r = requests.post("http://localhost:8000/hi", json={"who": "Mom"})  
>>> r.json()  
'Hello? Mom?'
```

HTTP Header

Finally, let's try passing the greeting argument as an HTTP header.

Example 3-12. Return the Greeting Header

```
from fastapi import FastAPI, Header
```

```
app = FastAPI()
```

```
@app.post("/hi")
```

```
def greet(who:str = Header()):
```

```
    return f"Hello? {who}?"
```

Let's test this one just with Httpie. It uses *name:value* to specify an HTTP header:

Example 3-13. Test With HTTPie

```
$ http -v localhost:8000/hi who:Mom
GET /hi HTTP/1.1
Accept: */\*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1
who: Mom
```

```
HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Mon, 16 Jan 2023 05:14:46 GMT
server: uvicorn
```

```
"Hello? Mom?"
```

FastAPI converts HTTP header keys to lowercase, and dash (-) to underscore (_). So you could print the value of the HTTP User-Agent header like this:

Example 3-14. Return the User-Agent Header

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.post("/agent")
def get_agent(user_agent:str = Header()):
    return user_agent
```

Example 3-15. Test the User-Agent Header With HTTPie

```
$ http -v localhost:8000/agent
GET /agent HTTP/1.1
```

```
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1

HTTP/1.1 200 OK
content-length: 14
content-type: application/json
date: Mon, 16 Jan 2023 05:21:35 GMT
server: uvicorn

"HTTPie/3.2.1"
```

Multiple Request Data

You can use more than one of these methods in the same path function. That is, you can get data from the URL, query parameters, the HTTP body, HTTP headers, cookies, and so on. And you can write your own dependency functions that process and combine them in special ways, such as for pagination or authentication. You'll see some of these in chapter 6, and various chapters in part 3.

Which Method is Best?

- When passing arguments in the URL, it's standard practice to follow RESTful guidelines.
- Query strings are usually used to provide optional arguments, like pagination.
- The body is usually used for larger inputs, like whole or partial models.

In each case, if you provide type hints in your data definitions, your arguments will be automatically type-checked by Pydantic. This ensures that they're both present and correct.

HTTP Responses

By default, FastAPI converts whatever you return from your endpoint function to JSON — the HTTP response has a header line `Content-type: application/json`. So, although the `greet()` function initially returned the string `"Hello? world?"`, FastAPI converted it to JSON. This is one of the defaults chosen by FastAPI to streamline API development.

In this case, the Python string `"Hello? world?"` is converted to its equivalent JSON string `"Hello? world?"`, which is the same darn string. But anything that you return is converted to JSON, whether built-in Python types or pydantic models. You'll see these in the coming chapters.

Automated Documentation

Convince your browser to visit the URL `http://localhost:8000/docs`.

You'll see something that starts like Figure 3-1 (I've cropped the following screen shots to emphasize particular areas):

default ^

POST /hi Greet ✓

Schemas ^

HTTPValidationError >

ValidationError >

Figure 3-1. Generated documentation page

Where did that come from?

FastAPI generates an OpenAPI specification from your code, and includes this page to display *and test* all of your endpoints. This is just one ingredient of its secret sauce.

Click on the down arrow on the right side of the green box to open it for testing:

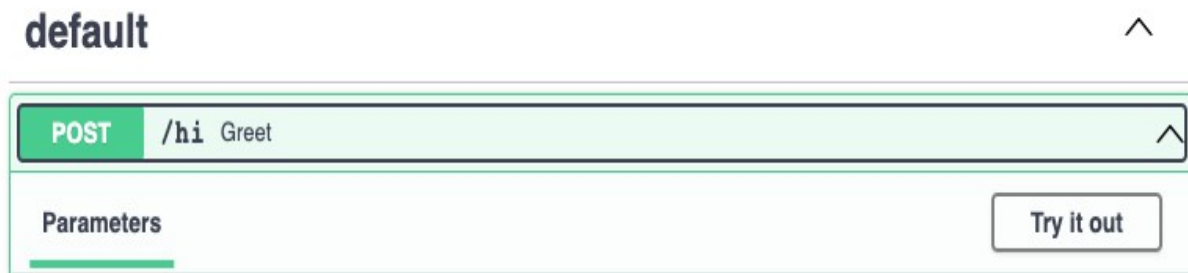


Figure 3-2. Open documentation page

Click that **Try it out** button on the right. Now you'll see an area that will let you enter a value in the body section:

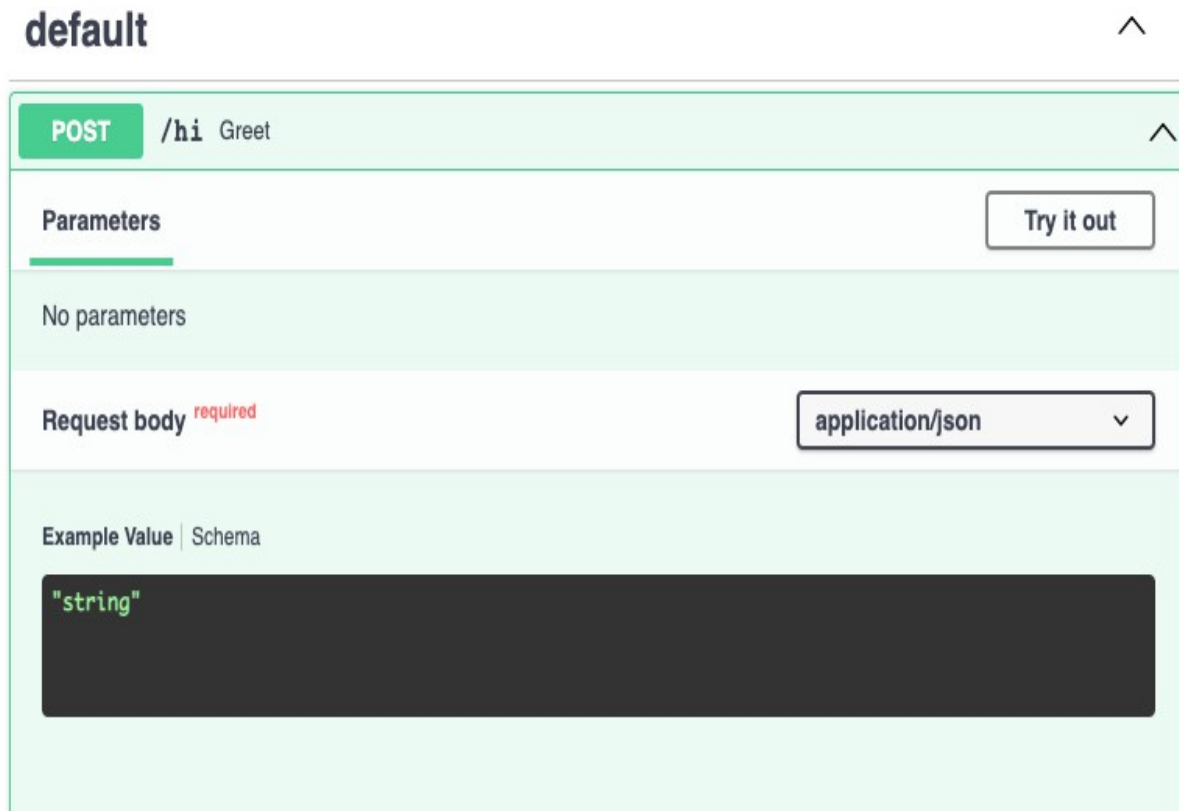


Figure 3-3. Data entry page

Click that "string". Change it to "Cousin Eddie" (keep the double quotes around it). Then click the bottom blue **Execute** button.

Now look at the **Responses** section below the **Execute** button:

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/hi' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '"Cousin Eddie"'
```

Request URL

```
http://localhost:8000/hi
```

Server response


Code	Details
200	<div><div>Response body</div><pre>"Hello? Cousin Eddie?"</pre><div> <button>Download</button></div></div> <div><div>Response headers</div><pre>content-length: 22 content-type: application/json date: Fri, 08 Jul 2022 10:34:17 GMT server: uvicorn</pre></div>

Figure 3-4. Response page

The **Response body** box shows that Cousin Eddie turned up.

So, this is yet another way to test the site (besides the earlier examples using the browser, Httpie, and Requests).

By the way, as you can see in the **Curl** box of the **Responses** display, using Curl for command-line testing instead of Httpie would have required more typing. Httpie’s automatic JSON encoding helps here.

TIP

This automated documentation is actually a big furry deal. As your web service grows to hundreds of endpoints, a documentation and testing page that's always up to date is very helpful. Chapter 8 has more details on other uses of OpenAPI.

Complex Data

These examples only showed how to pass a single string to an endpoint. Many endpoints, especially GET or DELETE ones, may need no arguments at all, or only a few simple ones, like strings and numbers. But when creating (POST) or modifying (PUT or PATCH) a resource, we usually need more complex data structures. Chapter 5 shows how FastAPI uses Pydantic and data models to implement these cleanly.

Review

In this chapter, we used FastAPI to create a website with a single endpoint. We tested it with three different web clients: a web browser, the httpie text program, and the requests Python package. Starting with a simple GET call, we passed arguments to it via the URL *path*, a query *parameter*, and an HTTP *header*. Then, the HTTP *body* was used to send data to a POST endpoint. Finally, an automatically-generated form page provided both documentation and live forms for a fourth test client.

This FastAPI overview will be expanded in chapter 9.

[1](#) Watch your spell checker.

[2](#) Technically, we're not creating anything, so a POST isn't kosher, but if the RESTful Overlords sue us, then hey, check out the cool courthouse.

Chapter 4. The Web Parts: Concurrency, Async, and Starlette

Starlette is a lightweight ASGI framework/toolkit, which is ideal for building async web services in Python.

Tom Christie

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

The previous chapter was a brief introduction to the first things a developer would encounter on writing a new FastAPI application. This chapter emphasizes FastAPI’s underlying Starlette library, particularly its support of *async* processing. After an overview of multiple ways of “doing more things at once” in Python, you’ll see how its newer *async* and *await* keywords have been incorporated into Starlette and FastAPI.

Starlette

Much of FastAPI's web code is based on the [Starlette](#) package, created by Tom Christie. It can be used as a web framework in its own right, or as a library for other frameworks, such as FastAPI. Like any other web framework, Starlette handles all the usual HTTP request parsing and response generation. It's similar to [Werkzeug](#), the package that underlies Flask.

But its most important feature is its support of the modern Python asynchronous web standard: [ASGI](#). Until now, most Python web frameworks (like Flask and Django) have been based on the traditional synchronous [WSGI](#) standard. Because web applications so frequently connect to much slower code (e.g., database, file, and network access), ASGI avoids the blocking and “busy waiting” of WSGI-based applications.

As a result, Starlette and frameworks that use it are the fastest Python web packages, rivalling even Golang and NodeJS applications.

Types of Concurrency

Before getting into the details of the *async* support provided by Starlette and FastAPI, it's useful to know how many ways we can implement *concurrency*.

In *parallel* computing, a task is spread at the same time across multiple dedicated CPUs. This is common in “number crunching” applications like graphics and machine learning.

In *concurrent* computing, each CPU switches among multiple tasks. Some tasks take longer than others, and we want to reduce the total time needed. Reading a file or accessing a remote network service is literally thousands to millions of times slower than running calculations in the CPU.

Web applications do a lot of this slow work. How can we make web servers, or any servers, run faster? The following sections discuss some possibilities, from system-wide down to the focus of this chapter: FastAPI's implementation of Python's *async* and *await*.

Distributed and Parallel Computing

If you have a really big application — one that would huff and puff on a single CPU — you can break it into pieces and make the pieces run on separate CPUs in a single machine, or on multiple machines. There are many, many ways of doing this, and if you have such an application, you already know a number of them. Managing all these pieces is more complex and expensive. In this book, the focus will be on small- to medium-sized applications that could fit on a single box. And these applications can have a mixture of synchronous and asynchronous code, nicely managed by FastAPI.

Operating System Processes

An operating system (or OS, because typing hurts) schedules resources: memory, CPUs, devices, networks, and so on. Every program that it runs executes its code in one or more *processes*. The OS provides each process with managed, protected access to resources, including when they can use the CPU.

Most systems use *preemptive* process scheduling, not allowing any process to hog the CPU, memory, or any other resource. An OS continually suspends and resumes processes, according to its design and settings.

For developers, the good news is: not your problem! But the bad news (which usually seems to shadow the good) is: you can't do much to change it, even if you want to.

With CPU-intensive Python applications, the usual solution is to use multiple processes, and let the OS manage them. Python has a [multiprocessing](#) module for this.

Operating System Threads

You can also run *threads* of control within a single process. Python's [threading](#) package manages these.

Threads are often recommended when your program is I/O-bound, and multiple processes when you're CPU-bound. But threads are tricky to program, and can cause errors that are very hard to find. In *Introducing*

Python, I likened threads to ghosts wafting around in a haunted house: independent and invisible, detected only by their effects. Hey, who moved that candlestick?

Traditionally, Python kept the process-based and thread-based libraries separate. Developers had to learn the arcane details of either to use them. A more recent package called [concurrent.futures](#) is a higher-level interface that makes them easier to use.

As you'll see, you can get the benefits of threads more easily with the newer `async` functions. FastAPI actually also manages threads for normal synchronous functions (`def`, not `async def`) via *threadpools*.

Green Threads

A more mysterious mechanism is presented by *green threads* such as [greenlet](#), [gevent](#) and [eventlet](#). These are *cooperative* (not preemptive). They're similar to OS threads, but run in user space (i.e, your program) rather than in the OS kernel. They work by *monkey-patching* some standard Python functions to make concurrent code look like normal sequential code: they give up control when they would block waiting for I/O.

OS threads are “lighter” (use less memory) than OS processes, and green threads are lighter than OS threads. In some [benchmarks](#), all the `async` methods were generally faster than their `sync` counterparts.

NOTE

After you've read this chapter, you may wonder which is better: `gevent` or `asyncio`? I don't think there's a single preference for all uses. Green threads were implemented earlier (using ideas from the multiplayer game *Eve Online*). This book features Python's standard *asyncio*, which is used by FastAPI, is simpler than threads, and performs well.

Callbacks

Developers of interactive applications like games and graphic user interfaces are probably familiar with *callbacks*. You write functions and associate them with some event, like a mouse click, keypress, or time. The prominent Python package in this category is [Twisted](#). The name *Twisted* reflects the reality that callback-based programs are a bit “inside-out” and hard to follow.

Python Generators

Like most languages, Python usually executes code sequentially. When you call a function, Python runs it from its first line until its end or a return.

But in a Python *generator function*, you can stop and return from any point, *and go back to that point* later. The trick is the `yield` keyword.

In one Simpsons episode, Homer crashes his car into a deer statue, followed by three lines of dialogue. We can write a normal Python function to return these lines as a list, and have the caller iterate over them:

```
>>> def doh():
...     return ["Homer: D'oh!", "Marge: A deer!", "Lisa: A female deer!"]
...
>>> for line in doh():
...     print(line)
...
Homer: D'oh!
Marge: A deer!
Lisa: A female deer!
```

This works perfectly when lists are relatively small. But what if we’re grabbing all the dialogue from all the Simpsons episodes? Lists use memory.

Here’s how a generator function would dole the lines out:

```
>>> def doh2():
...     yield "Homer: D'oh!"
...     yield "Marge: A deer!"
```

```

... yield "Lisa: A female deer!"
...
>>> for line in doh2():
...     print(line)
...
Homer: D'oh!
Marge: A deer!
Lisa: A female deer!

```

Instead of iterating over a list returned by the plain function `doh()`, we're iterating over a *generator object* returned by the *generator function* `doh2()`. The actual iteration (`for ... in`) looks the same. Python returned the first string from `doh2()`, but kept track of where it was for the next iteration, and so on until the function ran out of dialogue.

Any function with a `yield` in it is a generator function. Given this ability to go back into the middle of a function and resume execution, the next section looks like a logical adaptation.

Python async, await, and asyncio

Python's [asyncio](#) features have been introduced over various releases. You're running at least Python 3.7, when the `async` and `await` terms became reserved keywords.

The following examples show a joke that's only funny when run asynchronously. Run both yourself, because the timing matters.

First, the unfunny:

```

>>> import time
>>>
>>> def q():
...     print("Why can't programmers tell jokes?")
...     time.sleep(3)
...
>>> def a():

```

```

... print("Timing!")
...
>>> def main():
...     q()
...     a()
...
>>> main()
Why can't programmers tell jokes?
Timing!

```

You'll see a three second gap between the question and answer. Yawn.
But the async version is a little different:

```

>>> import asyncio
>>>
>>> async def q():
...     print("Why can't programmers tell jokes?")
...     await asyncio.sleep(3)
...
>>> async def a():
...     print("Timing!")
...
>>> async def main():
...     await asyncio.gather(q(), a())
...
>>> asyncio.run(main())
Why can't programmers tell jokes?
Timing!

```

This time, the answer should pop out right after the question, followed by three seconds of silence — just as though a programmer is telling it. Ha ha! Ahem.

NOTE

I've used `asyncio.gather()` and `asyncio.run()` above, but there are multiple ways of calling async functions. When using FastAPI, you won't need to use these.

Python thinks:

- Execute `q()`. Well, just the first line right now.
- Okay, you lazy async `q()`, I've set my stopwatch and I'll come back to you in three seconds.
- In the meantime I'll run `a()`, printing the answer right away.
- No other `await`, so back to `q()`.
- Boring event loop! I'll sit here aaaand stare for the rest of the three seconds.
- Okay, now I'm done.

This example used `asyncio.sleep()` for a function that takes some time, much like a function that reads a file or accesses a website. You put `await` in front of the function that might spend most of its time waiting. And that function needs to have `async` before its `def`.

NOTE

If you define a function with `async def`, its caller must put an `await` before the call to it. And the caller itself must be declared `async def`, and *its* caller must `await` it, all the way up.

By the way, you can declare a function as `async` even if it doesn't contain an `await` call to another `async` function. It doesn't hurt.

FastAPI and Async

After that long field trip over hill and dale, let's get back to FastAPI, and why any of it mattered.

Because web servers spend a lot of time waiting, performance can be increased by avoiding some of that waiting — in other words, concurrency. Other web servers use many of the methods mentioned earlier — threads, *gevent*, and so on. One of the reasons that FastAPI is one of the fastest Python web frameworks is its incorporation of async code, via the underlying *Starlette* package's ASGI support, and some of its own inventions.

NOTE

The use of `async` and `await` on their own does not make code run faster. In fact, it might be a little slower, from `async` setup overhead. The main use of `async` is to avoid long waits for I/O.

Now, let's look at our earlier web endpoint calls and see how to make them `async`.

FastAPI Path Functions

The functions that map URLs to code are called *path functions* in the FastAPI docs. I've also called them web endpoints, and you saw synchronous examples of them in Chapter 3. Let's make some `async` ones. Like those earlier examples, we'll just use simple types like numbers and strings for now. Chapter 5 introduces *type hints* and *Pydantic*, which we'll need to handle fancier data structures.

Example 4.1 below revisits our first FastAPI program from the previous chapter, and makes it asynchronous:

Example 4-1. A Shy Async Endpoint (greet_async.py)

```
from fastapi import FastAPI
import asyncio
```

```
app = FastAPI()
```

```
@app.get("/hi")
async def greet():
    await asyncio.sleep(1)
    return "Hello? World?"
```

To run that chunk of web code, you need a web server like uvicorn.

The first way is to run Uvicorn on the command line:

```
$ *uvicorn app:greet_async
```

The second is to call Uvicorn from inside the example code, when it's run as a main program instead of a module:

Example 4-2. A Shy Async Endpoint (greet_async_uvicorn.py)

```
from fastapi import FastAPI
import asyncio
import uvicorn
```

```
app = FastAPI()
```

```
@app.get("/hi")
async def greet():
    await asyncio.sleep(1)
    return "Hello? World?"
```

```
if __name__ == "__main__":
    uvicorn.run("main:app")
```

When run as a standalone program, Python names it "main". That if `__name__`... stuff is Python's way of only running it when called as a main program. Yes, it's ugly.

This will pause for one second before returning its timorous greeting. The only difference from a synchronous function that used the standard `sleep(1)` function: the web server can handle other requests in the meantime with the `async` example.

Using `asyncio.sleep(1)` fakes some real-world function that might take one second, like calling a database call or downloading a web page. Later chapters will show examples of such calls from this router layer to the service layer, and from there to the data layer, actually spending that wait time on real work.

FastAPI calls this `async greet()` path function itself when it receives a GET request for the URL `/hi`. You don't need to add an `await` anywhere. But for any other `async def` function definitions that you make, the caller must put an `await` before each call.

NOTE

FastAPI runs an *async event loop* that coordinates the `async` path functions, and a *threadpool* for synchronous path functions. A developer doesn't need to know the tricky details, which is a great plus. For example, you don't need to run things like `asyncio.gather()` or `asyncio.run()`, as in the (standalone, non-FastAPI) joke example earlier.

Using Starlette Directly

FastAPI doesn't expose Starlette as much as it does Pydantic. Starlette is largely the machinery humming in the engine room, keeping things running smoothly.

But if you're curious, you could use Starlette directly to write a web application. Example 3.1 in the previous chapter might look like this:


```
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route

async def greeting(request):
    return JSONResponse('Hello? World?')

app = Starlette(debug=True, routes=[
    Route('/hi', greeting),
])
```

Run it with:

```
$ uvicorn app:starlette_hello
```

In my opinion, the FastAPI additions make web API development much easier.

Interlude: Cleaning the Clue House

You own a small (very small: just you) house cleaning company. You've been living on ramen, but just landed a contract that will let you afford much better ramen.

Your client bought an old mansion that was built in the style of the board game Clue, and wants to host a character party there very soon. But the place is an incredible mess. If Marie Kondo saw the place, she might:

1. Scream
2. Gag
3. Run away
4. All of the above

There's a speed bonus in your contract. How can you clean the place thoroughly, in the least amount of elapsed time? The best approach would have been to have more CPUs (Clue Preservation Units), but you're it.

So you can:

1. Do everything in one room, then everything in the next, etc.
2. Do a specific task in one room, then the next, etc. Like polishing the silver in the Kitchen and Dining Room, or the balls in the Billiard Room.

Would your total time for these approaches differ? Maybe. But it might be more important to consider if you have to wait an appreciable time for some step. An example might be underfoot: after cleaning rugs and waxing floors, they might need to dry for hours before moving furniture back onto them.

So, your plan for each room is:

1. Clean all the static parts (windows, etc.).
2. Move all the furniture from the room into the Hall.
3. Remove years of grime from the rug and/or hardwood floor.
4. Either:
 1. Wait for the rug or wax to dry, but wave your bonus goodbye.
 2. Go to the next room now, and repeat. After the last room, move the furniture back into the first room, and so on.

The waiting-to-dry approach is the synchronous one, and it might be best if time isn't a factor and you need a break. The second is async, and saves the waiting time for each room.

Let's assume you chose the async path, because money. You got the old dump to sparkle and received that bonus from your grateful client. His later party turned out to be a great success, except:

1. One confused guest came as Mario.
2. You overwaxed the dance floor in the Ball Room, and a tipsy Professor Plum skated about in his socks, until he sailed into a table and spilled champagne on Miss Scarlet.

Morals of this story:

- Requirements can be conflicting and/or strange.
- Estimating time and effort can depend on many things.
- Sequencing tasks may be as much art as science.
- You'll feel great when it's all done. Mmm, ramen.

Review

After an overview of ways of increasing concurrency, this chapter expanded on functions that use the recent Python keywords `async` and `await`. It showed how FastAPI and Starlette handle both plain old synchronous functions and these new async funky functions.

The next chapter introduces the second leg of FastAPI: how Pydantic helps you define your data.

Chapter 5. The Data Parts: Type Hints and Pydantic

Data validation and settings management using Python type hints.

Fast and extensible, pydantic plays nicely with your linters/IDE/brain. Define how data should be in pure, canonical Python 3.6+; validate it with pydantic.

Samuel Colvin

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

FastAPI stands largely on a Python package called Pydantic. This uses *models* (Python object classes) to define data structures. These are heavily used in FastAPI applications, and are a real advantage when writing larger applications.

Type Hinting

It's time to learn a little more about Python type hints.

Chapter 2 mentioned that, in many computer languages, a variable points directly to a value in memory. This requires the programmer to declare its type, so the size and bits of the value can be determined. In Python, variables are just names associated with objects, and it's the objects that have types.

In normal programming, a variable is usually associated with the same object. If we associate a type hint with that variable, we can avoid some programming mistakes. So Python added type hinting to the language, in the standard module `typing`. The Python interpreter ignores the type hint syntax and runs the program as though it wasn't there. Then what's the point?

You might treat a variable as a string in one line, and forget later and assign it an object of a different type. Although compilers for other languages would complain, Python won't. The standard Python interpreter will catch normal syntax errors and runtime exceptions. but not mixing types for a variable. Helper tools like `Mypy` pay attention to type hints, and warn you about any mismatches.

Also, the hints are available to Python developers, who can write tools that do more than type error checking. The following sections describe how the `Pydantic` package was developed to address needs that weren't really obvious. Later, you'll see how its integration with `FastAPI` makes a lot of web development issues much easier to handle.

By the way, what do type hints look like? There's one syntax for variables, and another for function return values.

Variable type hints may include only the type:

```
name: type
```

or also initialize the variable with a value:

```
name: type = value
```

The *type* can be one of the standard Python simple types like `int` and `str`, or collection types like `tuple`, `list`, and `dict`.

```
thing: str = "yeti"
```

NOTE

Before Python 3.9, you need to import capitalized versions of these standard type names from the `typing` module.:

```
from typing import Str
thing: Str = "yeti"
```

Examples with initializations:

```
physics_magic_number: float = 1.0/137.03599913
hp_lovecraft_noun: str = "ichor"
exploding_sheep: tuple = "sis", "boom", "bah!"
responses: dict = {"Marco": "Polo", "answer": 42}
```

You can also include subtypes of collections:

```
name: dict[keytype, valtype] = {key1: val1, key2: val2}
```

The `typing` module has useful extras for subtypes, the most common being:

- `Any`: any type
- `Union`: any type of those specified, such as `Union[str, int]`.

NOTE

In Python 3.10 and up, you can say `type1 | type2` instead of `Union[type1, type2]`.

Examples:

```
from typing import Any
responses: dict[str, Any] = {"Marco": "Polo", "answer": 42}
```

or, a little more specific:

```
from typing import Union
responses: dict[str, Union[str, int]] = {"Marco": "Polo", "answer": 42}
```

or (Python 3.10 and up):

```
responses: dict[str, str | int] = {"Marco": "Polo", "answer": 42}
```

Notice that a type-hinted variable line is legal Python, but a bare variable line is not:

```
$ python
...
>>> thing0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name thing0 is not defined
>>> thing0: str
```

Also, incorrect type uses are not caught by the regular Python interpreter:

```
$ python
...
>>> thing1: str = "yeti"
>>> thing1 = 47
```

But they will be caught by Mypy. If you don't already have it, `pip install mypy`. Save those two lines above to a file called *stuff.py*[1](#), then try this:

```
$ mypy stuff.py
stuff.py:2: error: Incompatible types in assignment
(expression has type "int", variable has type "str")
Found 1 error in 1 file (checked 1 source file)
```

A function return type hint uses an arrow instead of a colon:

```
function(args) -> type:
```

Such as:

```
def get_thing() -> str:
    return "yeti"
```

You can use any type, including classes that you've defined, and combinations of them. You'll see that in a few pages.

Data Grouping

Often, we need to keep a related group of variables together, rather than passing around logs of individual variables. How do we integrate multiple variables as a group, and keep the type hints?

Let's leave behind our tepid greeting example from previous chapters, and start using richer data from now on. As in the rest of this book, we'll use examples of *cryptids* (imaginary creatures), and the (also imaginary) *explorers* who seek them.. Our initial creature definitions will only include string variables for *name*, *description*, and *location*.

Python's historic data grouping structures (beyond the basic int, string, and such) are:

- tuple: An immutable sequence of objects
- list: A mutable sequence of objects
- set: Mutable distinct objects
- dictionary: Mutable key:value object pairs (the key needs to be of an immutable type)

Tuples and lists only let you access a member variable by its offset, so you have to remember what went where:

```
>>> tuple_thing = ("yeti", "Abominable Snowman", "Himalayas")
>>> print("Name is", tuple_thing[0])
Name is yeti
```

```
>>> list_thing = ["yeti", "Abominable Snowman", "Himalayas"]
>>> print("Name is", list_thing[0])
Name is yeti
```

You can get a little more explanatory by defining names for the integer offsets:

```
>>> NAME = 0
>>> LOCATION = 1
>>> DESCRIPTION = 2
>>> tuple_thing = ("yeti", "Abominable Snowman", "Himalayas")
```

```
>>> print("Name is", tuple_thing[NAME])
Name is yeti
```

Dictionaries are a little better, giving you access by descriptive keys:

```
>>> dict_thing = {"name": "yeti",
... "description": "Abominable Snowman",
... "location": "Himalayas"}
>>> print("Name is", dict_thing["name"])
Name is yeti
```

Sets only contain unique values, so they're not very helpful for clustering different variables.

A *named tuple* is a tuple that gives you access by offset *or* name:

```
>>> from collections import namedtuple
>>> CreatureNamedTuple = namedtuple("CreatureNamedTuple",
... "name, location, description")
>>> namedtuple_thing = CreatureNamedTuple("yeti",
... "Abominable Snowman",
... "Himalayas")
>>> print("Name is", namedtuple_thing[0])
Name is yeti
>>> print("Name is", namedtuple_thing.name)
Name is yeti
```

NOTE

You can't say `namedtuple_thing["name"]`. It's a tuple, not a dict.

You can define a new Python class and add all the attributes you want to `self`. But you'll need to do a lot of typing just to define them:

```
>>> class CreatureClass():
...     def __init__(self, name: str, location: str, description: str):
...         self.name = name
...         self.location = location
...         self.description = description
...
>>> class_thing = CreatureClass("yeti",
...     "Abominable Snowman",
...     "Himalayas")
>>> print("Name is", class_thing.name)
Name is yeti
```

We'd like what other computer languages call a *record* or a *struct* (a group of names and values). A recent addition to Python is the *dataclass*. This example shows how all that `self` stuff disappears with dataclasses:

```
>>> from dataclasses import dataclass
>>>
>>> @dataclass
... class CreatureDataClass():
...     name: str
...     location: str
...     description: str
...
>>> dataclass_thing = CreatureDataClass("yeti",
...     "Abominable Snowman", "Himalayas")
>>> print("Name is", dataclass_thing.name)
Name is yeti
```

This is pretty good for the keeping-variables-together part. But we want more, so let's ask Santa for:

- A *union* of possible alternative types
- Missing/optional values
- Default values
- Data validation
- Serialization to and from formats like JSON

Pydantic

The [Pydantic](#) package, written by Samuel Colvin, is Santa's answer to our list above. It uses Python's standard type hinting that we've already been discussing, rather than some special syntax. It's central to FastAPI.

Models

It's tempting to use Python's built in data structures, especially dictionaries. But you'll inevitably find that dictionaries are a bit too "loose". Freedom comes at a price. You need to check *everything*:

- Is the key optional?
- If the key is missing, is there a default value?
- Does the key exist?
- If so, is the key's value of the right type?
- If so, is the value in the right range, or match a pattern?

Pydantic provides ways to specify any combination of these checks:

- Required vs. optional
- Default value if unspecified but required
- The data type or types expected
- Value range restrictions

- Other function-based checks if needed

A Simple Example

You've seen how to feed a simple string to a web endpoint via the URL, a query parameter, or the HTTP body. The problem is that usually you usually request and receive groups of data, of many types.

That's where Pydantic models first appear in FastAPI.

This initial example will use three files:

- *model.py* defines a Pydantic model
- *data.py* is a fake data source, defining an instance of a model
- *web.py* defines a FastAPI web endpoint that returns the fake data

For simplicity in this chapter, let's keep all the files in the same directory for now. In later chapters that discuss larger websites, I'll separate them into their respective layers.

First, define the *model* for a creature:

Example 5-1. model.py

```
from pydantic import BaseModel

class Creature(BaseModel):
    name: str
    location: str
    description: str

thing = Creature(location="Himalayas",
description="Abominable Snowman",
name="yeti")
print("Name is", thing.name)
```

The Creature class inherits from Pydantic's BaseModel. That : str part after name, location, and description is a type hint that each is a Python

string.

NOTE

In this example, all three fields are required. In Pydantic, if `Optional` is not in the type description, the field must have a value.

You can pass the arguments in any order if you include their names:

```
>>> thing = Creature(location="Himalayas",
... description="Abominable Snowman",
... name="yeti")
>>> print("Name is", thing.name)
Name is yeti
```

For now, we'll define a teeny source of data; in later chapters, we'll use databases. The following code shows the type hint `List[Creature]`, which tells Python that this is a list of Creature objects only.

Example 5-2. data.py

```
from model import Creature

_creatures: List[Creature] = [
    Creature(name="Yeti",
location="Himalayas",
description="Abominable Snowman"),
    Creature(name="Sasquatch",
location="North America",
description="Bigfoot; Yeti's Cousin Eddie in the New World"),
]

def get_creatures() -> List[Creature]:
    return _creatures
```

This code imported the *model.py* that we just wrote. It does a little data hiding by calling its list of Creature objects `_creatures`, and providing the function `get_creatures()` to return them.

For the final step, a file that defines a FastAPI web endpoint:

Example 5-3. web.py

```
from model import Creature
from fastapi import FastAPI

app = FastAPI()

@app.get("/creature")
def get_all() -> list[Creature]:
    from data import get_creatures
    return get_creatures()
```

Now fire up this one-endpoint server:

```
$ uvicorn creature:app
INFO: Started server process [24782]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

In another window, let's access it with the Httpie web client (try your browser or the Requests module if you like, too):

```
$ http http://localhost:8000/creature
HTTP/1.1 200 OK
content-length: 183
content-type: application/json
date: Mon, 12 Sep 2022 02:21:15 GMT
server: uvicorn

[
{
```

```
"description": "Abominable Snowman",
"location": "Himalayas",
"name": "Yeti"
},
{
"description": "Bigfoot, New World Cousin Eddie of the Yeti",
"location": "North America",
"name": "Sasquatch"
}
```

FastAPI and Starlette automatically converted the original Creature model object list into a JSON string. This is the default output format in FastAPI, so we didn't need to specify it.

Also, the window in which you originally started the Uvicorn web server should have printed a log line:

```
INFO: 127.0.0.1:52375 - "GET /creature HTTP/1.1" 200 OK
```

Getting Fancier: Type Validation

The previous section showed how to:

- Apply type hints to variables and functions
- Define and use a Pydantic model
- Return a list of models from a data source
- Return the model list to a web client, automatically converting the model list to JSON

Now, let's really put it to work validating data.

Try assigning a value of the wrong type to one or more of the Creature fields. Let's use a standalone test for this (Pydantic doesn't rely on any web code; it's a data thing).

Create this file and call it *test1.py*:

```
from model import Creature

dragon = Creature(
    name="dragon",
    location="Worldwide",
    description=["incorrect", "string", "list"]
)
```

Now test it:

```
$ python test1.py
Traceback (most recent call last):
  File "/Users/williamlubanovic/fastapi/ch5/test1.py", line 3, in <module>
    dragon = Creature(
  File "pydantic/main.py", line 342, in pydantic.main.BaseModel.__init__
pydantic.error_wrappers.ValidationError: 1 validation error for Creature
description
  str type expected (type=type_error.str)
```

This found that we assigned a list of strings to the description field, and it wanted a plain old string.

Fancy-Shmancier: Validating Values

Even if the value's type matches its specification in the Creature class, there may be more checks that need to pass. Some restrictions can be placed on the value itself.

- integer (conint) or float:
 - gt — Greater than

- lt — Less than
 - ge — Greater than or equal to
 - le — Less than or equal to
 - multiple_of — An integer multiple of a value
- string (constr):
 - min_length — Minimum character (not byte) length
 - max_length — Maximum character length
 - to_upper — Convert to upper case
 - to_lower — Convert to lower case
 - regex — Match a Python regular expression
- tuple, list, or set:
 - min_items — Minimum number of elements
 - max_items — Maximum number of elements

These are specified in the type parts of the model.

Let's ensure that the name field is always at least two characters long. Otherwise, "" (an empty string) is a valid string.

```
>>> from pydantic import BaseModel, constr
>>>
>>> class Creature(BaseModel):
...     name: constr(min_length=2)
...     location: str
...     description: str
...
>>> bad_creature = Creature(name="",
... location="your attic",
... description="it's a raccoon")
```

```

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "pydantic/main.py", line 342,
in pydantic.main.BaseModel.__init__
pydantic.error_wrappers.ValidationError:
1 validation error for Creature name
ensure this value has at least 2 characters
(type=value_error.any_str.min_length; limit_value=2)

```

That `constr` means a *constrained string*. You can also use the Pydantic `Field` specification:

```

>>> from pydantic import BaseModel, Field
>>>
>>> class Creature(BaseModel):
...     name: str = Field(..., min_length=2)
...     location: str
...     description: str
...
>>> bad_creature = Creature(name="!",
... location="your attic",
... description="it's a raccoon")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "pydantic/main.py", line 342,
in pydantic.main.BaseModel.__init__
pydantic.error_wrappers.ValidationError:
1 validation error for Creature name
ensure this value has at least 2 characters
(type=value_error.any_str.min_length; limit_value=2)

```

That `...` argument to `Field()` means that a value is required, and that there's no default value.

This is a minimal introduction to Pydantic. The main takeaway is that it lets you automate the validation of your data. You'll see how useful this is when

getting data from either the web or data ends.

Review

Models are the best way to define data that will be passed around in your web application. Pydantic leverages Python's *type hints* to define data models to pass around in your application.

Coming next: defining *dependencies* to separate specific details from your general code.

1 Do I have any detectable imagination? Hmm... No.

Chapter 6. Dependencies

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

One of the very nice design features of FastAPI is a technique called *dependency injection*. This phrase sounds technical and esoteric, but it’s a key aspect of FastAPI, and is surprisingly useful at many levels. This chapter looks at FastAPI’s built-in capabilities, as well as how to write your own.

What’s a Dependency?

A *dependency* is specific information that you need at some point. The usual way to get this information is to write code that gets it, right when you need it.

When you’re writing a web service, at some time you may need to:

- Gather input parameters from the HTTP request
- Validate inputs
- Check user authentication and authorization for some endpoints
- Look up data from some data source, often a database
- Emit metrics, logs, or tracking information

Web frameworks convert the HTTP request bytes to data structures, and you pluck what you need from them inside your router (web handling) functions as you go.

Problems with Dependencies

Getting what you want, right when you need it, and without external code needing to know how you got it, seems pretty reasonable. But it turns out that there are consequences:

- *Testing*: You can't test variations of your function that could look up the dependency differently.
- *Hidden dependencies*: Hiding the details means that something your function needs that could break when external things change.
- *Code duplication*: If your dependency is a common one (like looking up a user in a database, or combining values from an HTTP request) you might duplicate the lookup code in multiple functions.
- *OpenAPI visibility*: The automatic test page that FastAPI makes for you needs information from the dependency injection mechanism.

Dependency Injection

The phrase *dependency injection* is actually simpler than it sounds: pass any *specific* information that a function needs *into* the function. A traditional way to do this is to pass in a helper function, which you then call to get the specific data.

FastAPI Dependencies

FastAPI goes one step more: you can define dependencies as arguments to your function, and they are *automatically* called by FastAPI and pass in the *values* that they return. For example, a `user_dep` dependency could get the user's name and password from HTTP arguments, look them up in a database, and return a token that you use to track that user afterward. Your web handling function doesn't ever call this directly; it's handled at function call time.

You've already seen some dependencies, but didn't see them referred to as such: HTTP data sources like `Path`, `Query`, `Body`, and `Header`. These are actually functions or Python classes that dig the requested data from various areas in the HTTP request. They hide the details, like validity checks and data formats.

Why not write your own functions to do this? You could, but you would not have:

- Data validity checks
- Format conversions
- Automatic documentation

In many other web frameworks, you would do these checks inside your own functions. You'll see examples of this in Chapter 7, where I compare FastAPI with Python web frameworks like Flask and Django.

But in FastAPI, you can handle your own dependencies, much as the built-in ones do.

Writing a Dependency

In FastAPI, a dependency is something that's executed, so a dependency object needs to be of the type `Callable`, which includes functions and classes — things that you *call*, with parentheses and optional arguments.

Let's have an ultra simple example: a `user_dep()` dependency function that takes name and password string arguments, and just returns `True` if the user is valid. For our first version, let's have it return `True` for anything.

```
from fastapi import FastAPI, Depends, Params

app = FastAPI()

# the dependency function:
def user_dep(name: str = Params, password: str = Params):
    return {"name": name, "valid": True}

# the path function / web endpoint:
@app.get("/user")
def get_user(user: dict = Depends(user_dep)) -> dict:
    return user
```

Here, `user_dep()` is a dependency function. It acts like a FastAPI path function (it knows about things like `Params`, etc.), but doesn't have a path decorator above it. It's a helper, not a web endpoint itself.

The path function `get_user()` says that it expects an argument variable called `user`, and that variable will get its value from the dependency function `user_dep()`.

NOTE

In the arguments to `get_user()`, we could not have said `user = user_dep`, because `user_dep` is a Python function object. And we could not say `user = user_dep()`, because that would have called the `user_dep()` function when `get_user()` was *defined*, not when it's used. So we need that extra helper FastAPI `Depends()` function to call it just when it's wanted.

You can have multiple dependencies in your path function argument list.
(NOTE: more)

Dependency Scope

You can define dependencies to cover a single path function, a group of them, or the whole web application.

Single Path

In your *path function*, include an argument like this:

```
def pathfunc(name: depfunc = Depends(depfunc)):
```

or just:

```
def pathfunc(name: depfunc = Depends()):
```

name is whatever you want to call the value(s) returned by *depfunc*.

From the earlier example:

- *pathfunc* is `get_user()`
- *depfunc* is `user_dep()`
- *name* is `user`

Here's a very simple example that returns a fixed user name and a valid boolean:

Example 6-1. Return the User Dependency

```
from fastapi import FastAPI, Depends, Params
```

```
app = FastAPI()
```

```
# the dependency function:
def user_dep(name: str = Params, password: str = Params):
    return {"name": name, "valid": True}

# the path function / web endpoint:
@app.get("/user")
def get_user(user: dict = Depends(user_dep)) -> dict:
    return user
```

If your dependency function just checks something and doesn't actually return any values, you can also define the dependency in your path *decorator* (the preceding line, starting with a @):

```
@app.method(url, dependencies=[Depends(depfunc)])
```

Let's try that:

Example 6-2. Do a User Check Dependency

```
from fastapi import FastAPI, Depends, Params

app = FastAPI()

# the dependency function:
def check_dep(name: str = Params, password: str = Params):
    if not name:
        raise

# the path function / web endpoint:
@app.get("/check_user", dependencies=[Depends(check_dep)])
def check_user() -> bool:
    return True
```

Multiple Path Functions

Chapter 9 gives more details on how to structure a larger FastAPI application, including defining more than one *router* object under a top-level application, instead of attaching every endpoint to the top-level application.

```
from fastapi import FastAPI, Depends, APIRouter

router = APIRouter(..., dependencies=[Depends(_depfunc_)])
```

This will cause *depfunc* to be called for all path functions under router.

Global

When you define your top-level FastAPI application object, you can add dependencies to it that will apply to all of its path functions:

```
from fastapi import FastAPI, Depends

def depfunc1(...):
    pass

def depfunc2(...):
    pass

app = FastAPI(dependencies=[Depends(depfunc1), Depends(depfunc2)])

@app.get("/main")
def get_main():
    pass
```

In this case, I'm using `pass` to ignore the other details to show how to attach the dependencies.

Review

This chapter discussed dependencies and dependency injection — ways of getting the data you need when you need it, in a straightforward way.

Chapter 7. Comparisons

You don't need a framework. You need a painting, not a frame.

Klaus Kinski

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Preview

For developers who have used Flask, Django, or popular Python web frameworks, this chapter will point out FastAPI's similarities and differences. It will not go into every excruciating detail, because the binding glue wouldn't hold this book together. This can be useful if you're thinking of migrating an application from one of these to FastAPI, or just curious.

One of the first things you might like to know about a new web framework is how to get started, and a top-down way is by defining *routes* (mappings from URLs and method to functions). The next section compares how to do this with FastAPI and Flask, because they're more similar to one another than Django, and are more likely to be considered together for similar applications.

Flask

[Flask](#) calls itself a *micro-framework*. It provides the basics, and you download third-party packages to supplement it as needed. It's smaller than Django, and faster to learn when you're getting started.

Flask is synchronous, based on WSGI rather than ASGI. A new project called [quart](#) is replicating Flask and adding ASGI support.

Let's start at the top, showing how Flask and FastAPI define web routing.

(TO DO: include web server start code)

Path Route

At the top level, Flask and FastAPI both use a decorator to associate a route with a web endpoint. Let's duplicate example 3.2 (from back in chapter 3), which gets the person to greet from the URL path:

Example 7-1. FastAPI Path Example

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi/{who}")
def greet(who: str):
    return f"Hello? {who}?"
```

By default, FastAPI converts that `f"Hello? {who}?"` string to JSON and returns it to the web client.

Here's how Flask would do it:

Example 7-2. Flask Path Example

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/hi/<who>", methods=["GET"])
def greet(who: str):
    return jsonify(f"Hello? {who}?" )
```

Notice that the `who` in the decorator is now bounded by `<` and `>`. In Flask, the method needs to be included as an argument — unless it's the default, `GET`. So `methods=["GET"]` above could have been omitted here, but it never hurts to be explicit.

NOTE

Flask 2.0 supports the FastAPI-style decorators like `@app.get` instead of `app.route`.

The Flask `jsonify()` function converts its argument to a JSON string and returns it, along with the HTTP response header indicating that it's JSON. If you're returning a dict (not other data types), recent versions of Flask will automatically convert it to JSON and return it. Calling `jsonify()` explicitly works for all data types, including dicts.

Query Parameter Route

Now, let's repeat example 3.3, where `who` is passed as a query parameter (after the `?` in the URL):

Example 7-3. FastAPI Query Parameter Example

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

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

```
def greet(who):
```

```
    return f"Hello? {who}?"
```

The Flask equivalent is:

Example 7-4. Flask Query Parameter Example

```
from flask import Flask, request
```

```
app = Flask(__name__)
```

```

@app.route("/hi", methods=["GET"])
def greet():
    who: str = request.args.get("who")
    return jsonify(f"Hello? {who}?")

```

In Flask, we need to get request values from the request object. In this case, args is a dict containing the query parameters

Body Route

Now, copying old example 3.7:

Example 7-5. FastAPI Body Example

```

from fastapi import FastAPI

```

```

app = FastAPI()

```

```

@app.get("/hi")
def greet(who):
    return f"Hello? {who}?"

```

A Flask version would look like this:

Example 7-6. Flask Body Example

```

from flask import Flask, request

```

```

app = Flask(__name__)

```

```

@app.route("/hi", methods=["GET"])
def greet():
    who: str = request.json["who"]
    return jsonify(f"Hello? {who}?")

```

Flask stores JSON input in request.json.

Header Route

Finally, let's repeat example 3.11 here:

Example 7-7. FastAPI Header Example

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/hi")
def greet(who:str = Header()):
    return f"Hello? {who}?"
```

The Flask version is:

Example 7-8. Flask Header Example

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/hi", methods=["GET"])
def greet():
    who: str = request.headers.get("who")
    return jsonify(f"Hello? {who}?" )
```

As with query parameters, Flask keeps request data in the request object. This time, it's the headers dict attribute. The header keys are supposed to be case-insensitive.

(NOTE: verify; also, converts dash to underscore?)

Django

Django is bigger and more complex than Flask or FastAPI, targetting “perfectionists with deadlines”. Its built-in *ORM* (object relational mapper) is useful for sites with major database backends. It's more of a monolith than a toolkit. Whether the extra complexity and learning curve are justified depends on your application.

Although Django was a traditional WSGI application, version 3.0 added support for ASGI.

Unlike Flask and FastAPI, Django likes to define routes (associating URLs with web functions, which it calls *view functions*) in a single URLConf table, rather than using decorators. This makes it easier to see all your routes in one place, but makes it harder to see what URL is associated with a function when you're just looking at the function.

Other Web Framework Features

In the sections comparing the three frameworks above, I've mainly compared how to define routes. A web framework might be expected to help in other areas, too:

- *Forms*: All three packages support standard HTML forms.
- *Files*: All of these packages handle file uploads and downloads, including multipart HTTP requests and responses.
- *Templates*: A *template language* lets you mix text and code, and is useful for a *content-oriented* (HTML text with dynamically inserted data) website, rather than an API website. The best-known Python template package is [jinja2](#), and it's supported by Flask, Django, and FastAPI. Django also has its own [template language](#).

If you want to use networking methods beyond basic HTTP:

- *Server Sent Events*: Push data to a client as needed. Supported by FastAPI ([sse-starlette](#)), Flask ([flask-sse](#)), and Django ([EventStream](#)).
- *Queues*: Job queues, publish-subscribe, and other networking patterns and supported by external packages like ZeroMQ, Celery, Redis, and RabbitMQ.
- *WebSockets*: Supported by FastAPI (directly), Django ([Django Channels](#)), and Flask (third-party packages).

Databases

Flask and FastAPI do not include any database handling in their base packages, but database handling is a key feature of Django.

Your site's data layer might access a database at different levels:

- Direct SQL (PostgreSQL, SQLite)
- Direct NoSQL (Redis, MongoDB, ElasticSearch)
- An *ORM* (object relational mapper/manager) that generates SQL
- An *ODM* (object document/data mapper/manager) that generates NoSQL

For relational databases, [SQLAlchemy](#) is an excellent package that includes multiple access levels, from direct SQL up to an ORM. This is a common choice for Flask and FastAPI developers. The author of FastAPI has leveraged both SQLAlchemy and Pydantic for the [SQLModel](#) package, which is discussed more in chapter 12.

Django is often the framework choice for a site with heavy database needs. It has its own [ORM](#), and an automated [database admin page](#). Although some sources recommend letting nontechnical staff use this admin page for routine data management, be careful. I've seen one case where a non-expert misunderstood an admin page warning message, and the database needed to be manually restored from a backup.

Chapter 12 discusses FastAPI and databases in more depth.

Recommendations

For API-based services, FastAPI seems to be the best choice now. Flask and FastAPI are about equal in terms of getting a service up and running quickly. Django takes more time to understand, but provides many features of use for larger sites, especially those with heavy database reliance.

Other Python Web Frameworks

The current big three are Flask, Django, and FastAPI. Google “python web frameworks” and you’ll get a googleful of suggestions, which I won’t repeat here. A few that might not stand out in those lists, but that are interesting for one reason or another, include:

- [Bottle](#): a very minimal (single Python file) package, good for a quick proof of concept.
- [Starlite](#): similar to FastAPI — it’s based on ASGI/Starlette and Pydantic — but has its own opinions.
- [AIOHTTP](#): an ASGI client and server, with useful demo code.
- [Socketify](#): a new entrant that claims very high performance.

Review

Flask and Django are the most popular Python web frameworks, although FastAPI’s popularity is growing faster. All three handle the basic web server tasks, with varying learning curves. FastAPI seems to have a cleaner syntax for specifying routes, and its support of ASGI allows it to run faster than its competitors in many cases.

Coming next: a brief overview of how FastAPI uses OpenAPI.

About the Author

Bill Lubanovic has been a developer for over forty years, specializing in Linux, the web, and Python. He coauthored the O'Reilly book *Linux System Administration*, and wrote both editions of *Introducing Python*. He discovered FastAPI a few years ago, and with his team used it to rewrite a large biomedical research API. The experience was so positive that they've adopted it for all new projects.

Table of Contents

[Preface](#)

[Conventions Used in This Book](#)

[Using Code Examples](#)

[O'Reilly Online Learning](#)

[How to Contact Us](#)

[I. What's New?](#)

[1. The Modern Web](#)

[Preview](#)

[Services and APIs](#)

[Kinds of APIs](#)

[HTTP](#)

[REST\(ful\)](#)

[JSON and API Data Formats](#)

[JSON:API](#)

[GraphQL](#)

[Concurrency](#)

[Layers](#)

[Data](#)

[Review](#)

[2. Modern Python](#)

[Preview](#)

[Tools](#)

[Getting Started](#)

[Python Itself](#)

[Package Management](#)

[Virtual Environments](#)

[Poetry](#)

[Source Formatting](#)

[Testing](#)

[Source Control and Continuous Integration
\(CI\)](#)

[Web Tools](#)

[APIs and Services](#)

[Variables Are Names](#)

[Type Hints](#)
[Data Structures](#)
[Web Frameworks](#)

[Django](#)
[Flask](#)
[FastAPI](#)

[Review](#)

[II. A FastAPI Tour](#)

[3. Overview](#)

[Preview](#)
[What Is FastAPI?](#)
[A FastAPI Application](#)
[HTTP Requests](#)

[URL Path](#)
[Query Parameters](#)
[Body](#)
[HTTP Header](#)

[Multiple Request Data](#)
[Which Method is Best?](#)
[HTTP Responses](#)
[Automated Documentation](#)
[Complex Data](#)
[Review](#)

[4. The Web Parts: Concurrency, Async, and Starlette](#)

[Preview](#)
[Starlette](#)
[Types of Concurrency](#)

[Distributed and Parallel Computing](#)
[Operating System Processes](#)
[Operating System Threads](#)
[Green Threads](#)
[Callbacks](#)
[Python Generators](#)
[Python async, await, and asyncio](#)

[FastAPI and Async](#)

[FastAPI Path Functions](#)

[Using Starlette Directly](#)

[Interlude: Cleaning the Clue House](#)
[Review](#)

[5. The Data Parts: Type Hints and Pydantic](#)

[Preview](#)
[Type Hinting](#)
[Data Grouping](#)
[Pydantic](#)
[Models](#)
[A Simple Example](#)
[Getting Fancier: Type Validation](#)
[Fancy-Shmancier: Validating Values](#)
[Review](#)

[6. Dependencies](#)

[Preview](#)
[What's a Dependency?](#)
[Problems with Dependencies](#)
[Dependency Injection](#)
[FastAPI Dependencies](#)
[Writing a Dependency](#)
[Dependency Scope](#)
[Single Path](#)
[Multiple Path Functions](#)
[Global](#)

[Review](#)

[7. Comparisons](#)

[Preview](#)
[Flask](#)
[Path Route](#)
[Query Parameter Route](#)
[Body Route](#)
[Header Route](#)

[Django](#)
[Other Web Framework Features](#)
[Databases](#)
[Recommendations](#)
[Other Python Web Frameworks](#)
[Review](#)