O'REILLY®

FastAPI

Modern Python Web Development

Early Release
RAW & UNEDITED

Bill Lubanovic

FastAPI

FIRST EDITION

Modern Python Web Development

Bill Lubanovic

Beijing • Boston • Farnham • Sebastopol • Tokyo

FastAPI

by Bill Lubanovic

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* .

Editors: Corbin Collins and Amanda Quinn

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2023: First Edition

Revision History for the First Edition

- 2022-09-14: First Early Release

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

978-1-098-13550-8

[FILL IN]

Dedication

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

Preface

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/oreillymedia/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: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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.

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*

Acknowledgments

Part I. What's New?

The world has benefited greatly from the invention of the World Wide Web by Sir Tim Berners-Lee[1], 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 digress[2].

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.

---

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

2 Not 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

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 are 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 the *backend* (database access) and middle (business logic), while JavaScript or mobile apps provide rich *frontends*. 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].

The frontends and backends 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.

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. 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.

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.

But the most common way to perform APIs on the Internet, and the focus of this book, is through the web's basic protocol: HTTP.

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

(NOTE: figure here)

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.

(NOTE: discussion of JSON:API vs RESTful)

GraphQL

RESTful interfaces can be cumbersome for some purposes. Facebook designed *GraphQL* (Graph Query Language) to specify more flexible service queries.

(NOTE: brief comparison of how GraphQL requests and responds)

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 days[3], 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.

Instead, *asynchronous* systems provide an *event loop*: requests are sent and noted, but we don’t wait for slow 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.

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* model[4]. This actually isn't terribly new. Terms differ[5], 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

(NOTE: image here)

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 Routing 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:

(NOTE: boxes image here)

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.

(NOTE: annotated arrows image here)

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

- Client ⇔ Router: RESTful HTTP with JSON
- Router ⇔ Service: Models
- Service ⇔ Data: Models
- Data ⇔ 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

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 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 at least Python 3.7. This includes newer features like type hints and asyncio, which are core requirements for FastAPI. 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.

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, 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 envirnment, 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 there are simpler tools like nose, the industrial-strength Python test package 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. Ths 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

(NOTE: figure here)

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.

(NOTE: figure here)

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).

(NOTE: figure here)

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.

(NOTE: figure here)

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.

(NOTE: figure here)

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.

Concurrency

For the longest time, Python was strictly synchronous — one thing ran at a time, for as long as it needed. If you called something slow, like file or network I/O, then your program *blocked* (sat there and waited until the bytes came back). If you wanted to overlap processing — have some *concurrency* — Python has had many alternatives. You'll see details about these in Chapter 4.

Data Structures

Often, we need to keep a related group of variables together. 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)

Other standard Python data structures can provide something like what other computer languages call a *record* (a group of names and values):

- Namedtuple
- Dataclass

Actually, 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.

And we want more, so while we're at it, let's ask Santa for:

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

As you'll see in chapter 5, the third-party package pydantic. was chosen by FastAPI to do all of these things.

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 Gatewway Interface) is a synchronous Python standard specification to connect application code to web servers.

As mentioned in the previous chapter, 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:

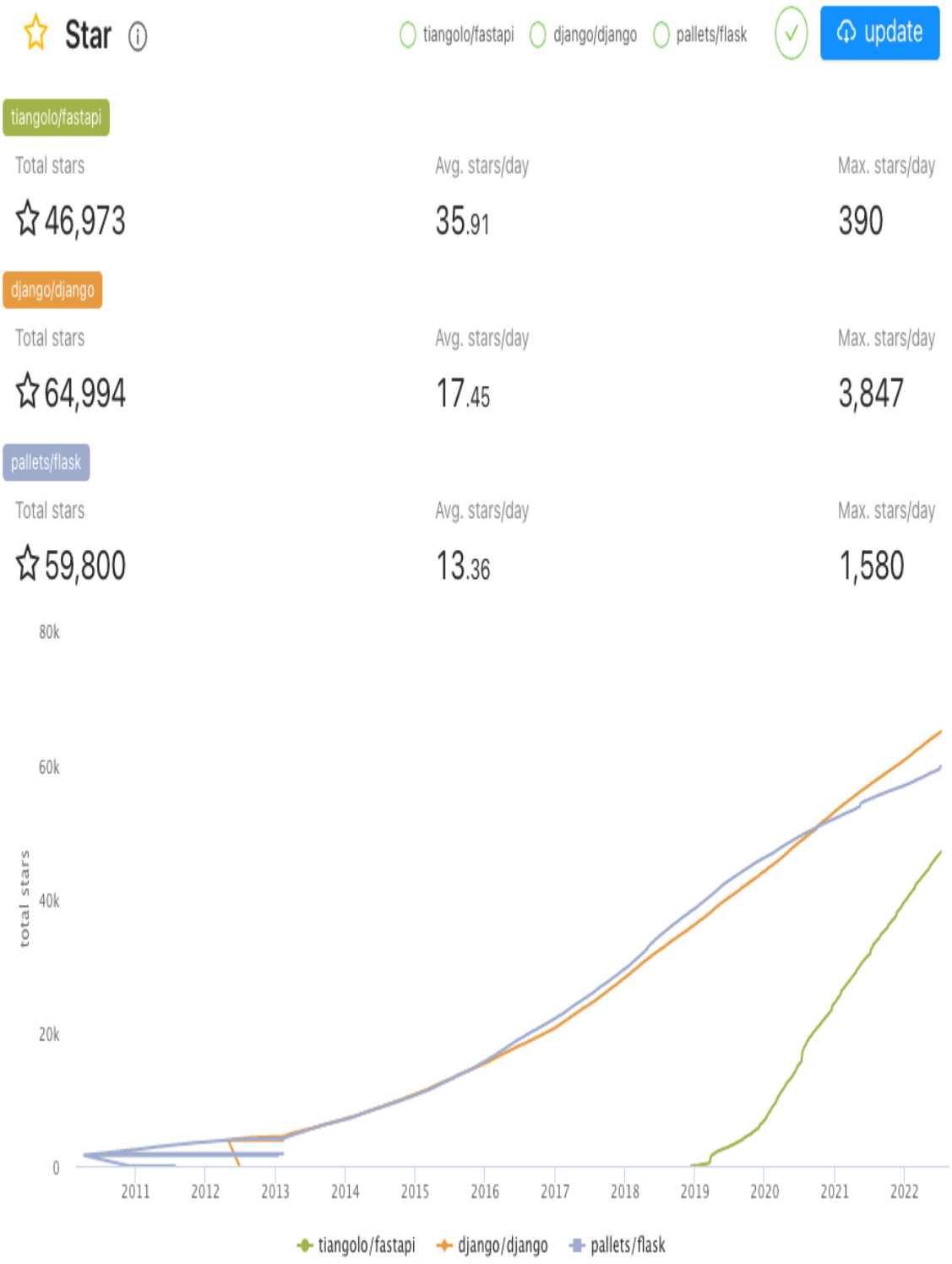

Figure 2-1. Github Stars

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

This part is a whatever-thousand foot/meter view of FastAPI — more like a drone than a spy satellite. It covers the basics quickly, but stays above the water line to avoid drowning you in details. It's meant to provide context for the things to come.

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

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 golang 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?"
```

How do we test this?

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
```

(NOTE: figure here)

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 output (response headers and body), so you can see everything that you're getting back.

HTTP Requests

That example included only one specific request: the endpoint `/hi` for the server `localhost` on port `8000`.

Web requests squirrel different data in different parts of an HTTP request. 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

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 three different ways to pass this new parameter:

- In the URL *path*
- A *query* parameter, after the `?` in the URL
- In the HTTP *body*

URL Path

Edit *hello.py*:

Example 3-2. Using a Path Parameter

```
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.

> **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:

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?'
```

Query Parameters

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

Example 3-3. Using a 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.

Test this one in your browser:

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

or 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 `==`):

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

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

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:

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi", params=
```

```
{"who": "Mom"})
>>> r.json()
'Hello? Mom?'
```

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 example[2].

Example 3-4. Using a Body Parameter

```
from fastapi import FastAPI, Body

app = FastAPI()

@app.post("/hi")
def greet(who:str = Body(embed=True)):
    print(who)
    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):

```
$ 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:

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

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):

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:

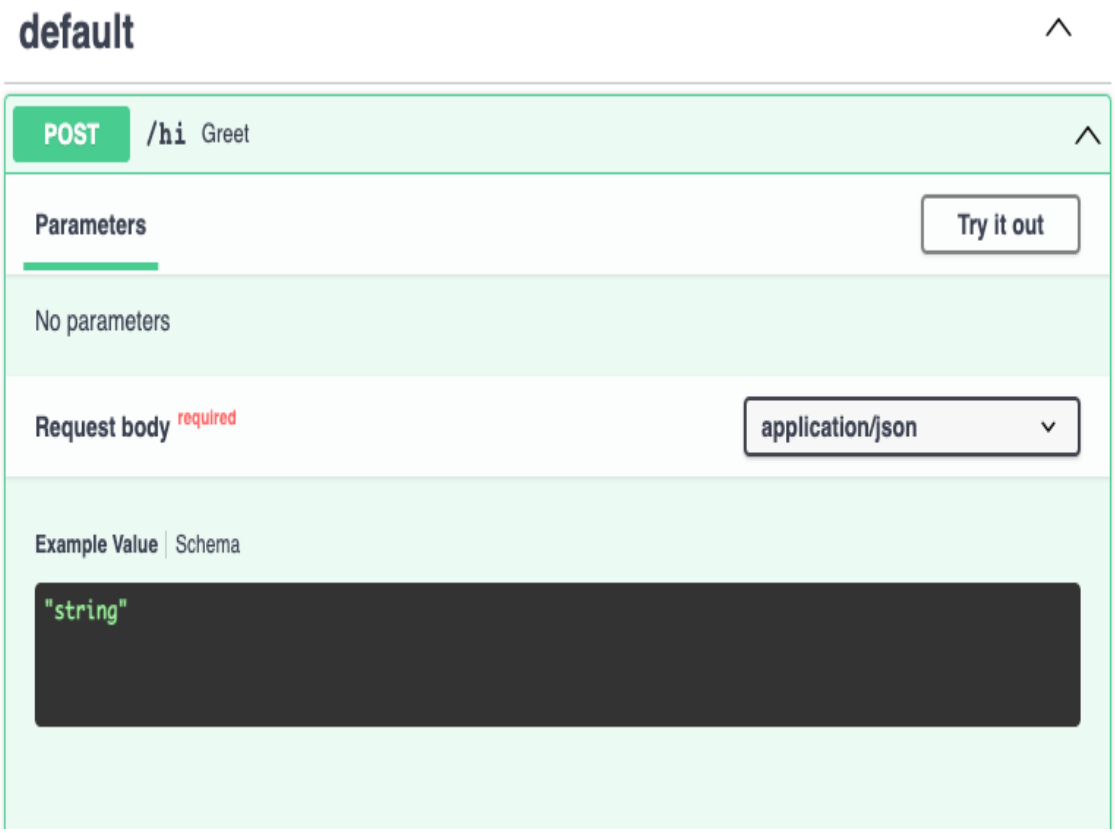

Figure 3-3. Data entry page

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:

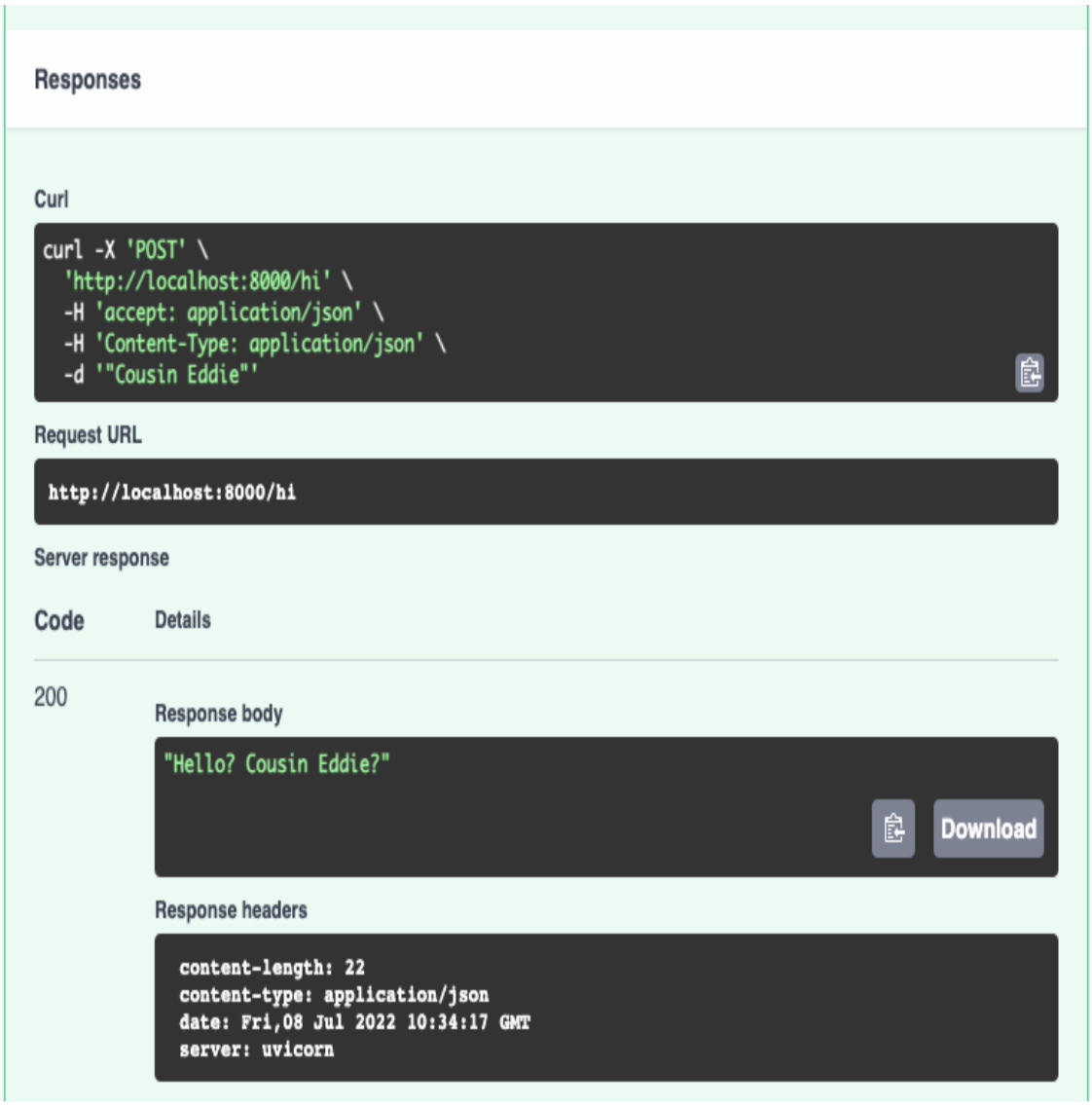

Figure 3-4. Response page

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.

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 and a query parameter. Then, the HTTP body was used to send data to a `POST` endpoint. Finally, an automatically-generated 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

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.

(NOTE: figure here)

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 machiner 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. This 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.

(NOTE: more)

Operating System Threads

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

(NOTE: more)

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 operating system 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 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 familar 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 Simpson's episode, Homer crashes his car into a deer statue, followed by three lines of dialogue. We'd normally 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 Simpson's 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 a 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. In this book, we assume 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 web site. 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 spent 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.

Let's revisit our first FastAPI program from the previous chapter, and make it asynchronous:

Example 4-1. A Shy Async Endpoint

```
from fastapi import FastAPI
import asyncio

app = FastAPI()

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

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 eaxmple, you don't need to run things like `asyncio.gather()` or `asyncio.run()`, as in the (standalone, non-FastAPI) joke example earlier.

(NOTE: more)

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.

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.

About the Author

Bill Lubanovic lives with his family and cats in the Sangre de Sasquatch mountains of Minnesota.