O'REILLY

FastAPI

Modern Python Web Development

Early Release

RAW & UNEDITED

Bill Lubanovic

FastAPI

Modern Python Web Development

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
- 2023-05-04: Third Release
- 2023-06-28: Fourth Release

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

978-1-098-13550-8

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.) will be available in the future.

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.

O'Reilly Online Learning

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*

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:

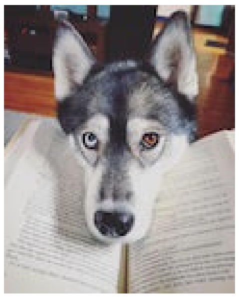


Figure I-1. FastAPI: Modern Pooch Woof Development

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

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 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 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 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 until it responds.

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 mush 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.

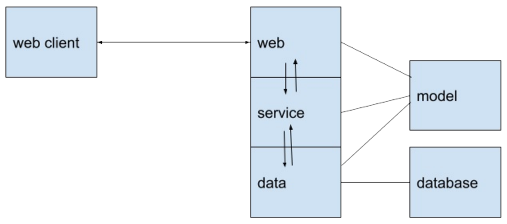
Layers

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* model[4]. This actually isn't terribly new. Terms differ[5], but for this book I'm using the following simple separation and terms:

- *Web*: 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

- *Model*: Data definitions shared by all layers



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 Web 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 alternative text (or *CLI*, command line interface) clients that could be built alongside the Web layer. Python Web 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.

The Model box isn’t an actual layer, but a source of data definitions shared by the layers. This isn’t needed if you’re passing built-in Python data structures among them. As you will see, FastAPI’s inclusion of Pydantic enables the definition of data structures with many useful features.

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.

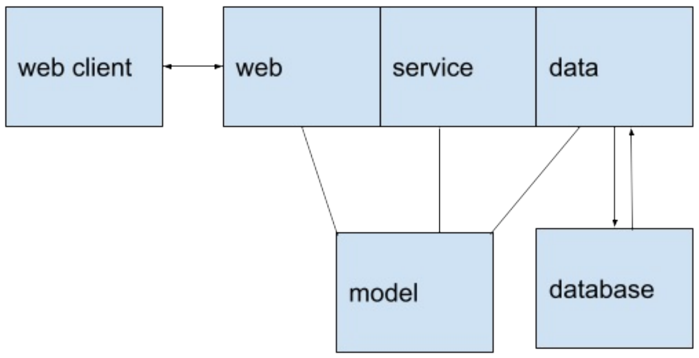
Follow one rule from *Ghostbusters*: *Don’t cross the streams*. That is, don’t let web details leak out of the Web layer, or database details out of the Data layer.

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 Web 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 ⇔ Web: RESTful HTTP with JSON
- Web ⇔ 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 11 looks at “auth” requirements, and how to implement them — as an extra layer between Web 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 Example 2-1:

Example 2-1. The Python program that goes like this: 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 Example 2-2:

Example 2-2. Test this.py

```
$ 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.

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 12. 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. 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 12 (Testing) and 15 (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, you 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 on Python and data structures 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. As the saying goes, don’t reinvent the wheel — unless you can’t get a round one.

WSGI (the Web Server Gatewway Interface) is a synchronous Python standard specification to connect application code to web servers. Traditional Python web frameworks are all built on WSGI. But synchronous communication may mean busy waiting for something that’s *much* slower than the CPU, like a disk or network. Then you’ll look for some better *concurrency*. Concurrency has become more important in recent years. As a result, the Python *ASGI* (Asynchronous Standard Gateway Interface) specification was developed. Chapter 4 talks about this.

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

After meeting other suitors at the ball, we finally encounter the intriguing 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, and is growing faster. A 2022 comparison shows that it may pass them at some point.

NOTE

As of May 2023, the github star counts are:

- Django: 70.3k
- Flask: 62.8k
- FastAPI: 57.4k

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 III.

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. FastAPI Tour

> *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 overview of FastAPI's main features, with emphasis on the first things that you'll want to know — how to handle web requests and responses.

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 and validation
- 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.

Let's shadow an introverted web developer in Example 3-1 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?"
```

Things to notice include:

- `app` is the top-level FastAPI object that represents the whole web application.
- `@app.get("/hi")` is a *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 a *path function* — the 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.

The next step is to run this web applucation in a web server. FastAPI itself does not include a web server, but recommends `Uvicorn`. There are two ways to start Uvicorn and the FastAPI web application:

Externally, via the command line, in Example 3-2:

Example 3-2. Start Uvicorn with the command line

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

The `hello` above refers to the *hello.py* file, and the `app` is the FastAPI variable name within it.

Or internally, in the application itself, in Example 3-3:

Example 3-3. Start Uvicorn internally

```
from fastapi import FastAPI

app = FastAPI()

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

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("hello:app", reload=True)
```

In either case, that `reload` tells Uvicorn to restart the web server if *hello.py* changes. In this chapter, we're going to do that a lot.

Either case will use port 8000 on your machine (named `localhost`) by default. Both the external and internal methods have `host` and `port` arguments if you'd prefer something else.

Now the server has a single endpoint (*/hi*) and is ready for requests.

Let's test with multiple web 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 or Httpx, 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
```

Examples 3-4 through 3-7 show different ways to test the webserver's brand new `/hi` endpoint.

Example 3-4. Test `/hi` in the browser

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

Example 3-5. Test `/hi` with Requests

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

Example 3-6. Test `/hi` with Httpx, which 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.

Example 3-7. Test `/hi` with 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?"
```

Use the `-b` argument in Example 3-8 to skip the response headers and onlh print the body:

Example 3-8. Test `/hi` with Httpie, only printing the response body

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

Example 3.9 gets the full request headers as well as the response with `-v`:

Example 3-9. Test `/hi` with Httpie and get everything

```
$ 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?"
```

Some examples in this book will show the default Httpie output (response headers and body), and some just the body.

HTTP Requests

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

Web requests squirrel data in different parts of an HTTP request, and FastAPI lets you access them smoothly. From the sample request above, Example 3-10 shows the HTTP request that the `http` command sent to the web server:

Example 3-10. An HTTP request

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

It contains:

- The verb (`GET`) and path (`/hi`).
- Any *query parameters* (stuff after any `?`; in this case, none).
- Other HTTP headers.
- No request body content.

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 will be discussed as we go along, and expanded 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* in Example 3-11:

Example 3-11. Return the greeting path

```
from fastapi import FastAPI

app = FastAPI()

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

Once you save this change from your editor, Uvicorn should restart. (Otherwise, we'd create *hello2.py*, etc. and rerun Uvicorn each time.) If you have a typo, keep trying until you fix it, and Uvicorn won't give you a hard time.

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 a Python 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.

In Examples 3-12 through 3-14, test this modified endpoint with the various methods discussed earlier:

Example 3-12. Test `/hi/Mom` in the browser

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

Example 3-13. Test `/hi/Mom` with 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?"
```

Example 3-14. Test `/hi/Mom` with 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.

The response in each case is the JSON string (with single or double quotes, depending on which test client you used) `"Hello? Mom?"`.

Query Parameters

Query parameters are the *name=value* strings after the `?` in a URL, separated by `&` characters. Edit *hello.py* again in Example 3-15:

Example 3-15. 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. Test with Examples 3-16 and 3-17:

Example 3-16. Test with your browser

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

Example 3-17. Test with HTTPie

```
$ http -b localhost:8000/hi?who=Mom
"Hello? Mom?"
```

In Example 3-18, you can call Httpie with a query parameter argument (note the `==`):

Example 3-18. Test with HTTPie and params

```
$ http -b localhost:8000/hi who==Mom
"Hello? 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.

Examples 3-19 and 3-20 show the same alternatives for Requests:

Example 3-19. Test with Requests

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

Example 3-20. Test with Requests and params

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

In each case, I provided the `"Mom"` string in a new way, and got it to the path function and through to 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, in Example 3-21 let’s change the endpoint from a `GET` to a `POST`footnote:[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.].

Example 3-21. 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 in Example 3-22, using `-v` to show the generated request body (and note the single `=` parameter to indicate JSON body data):

Example 3-22. 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 in Example 3-23, which uses its `json` argument to pass JSON-encoded data in the request body:

Example 3-23. 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 in Example 3-24:

Example 3-24. Return the greeting header

```python
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 in Example 3-25. It uses *name:value* to specify an HTTP header:

Example 3-25. 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 in Examples 3-26 and 3-27:

Example 3-26. Return the User-Agent header

```python
from fastapi import FastAPI, Header

app = FastAPI()

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

Example 3-27. 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.

Status Code

By default, FastAPI returns a 200 status code; exceptions raise 40x codes.

In the path decorator, specify the HTTP status code that should be returned if all goes well (exceptions will generate their own codes and override it). Add the code from Example 3-28 somewhere in your *main.py* (just to avoid showing the whole file again and again), and test it with Example 3-29:

Example 3-28. Specify HTTP status code

```
@app.get("/happy")
def happy(status_code=200):
    return ":)"
```

Example 3-29. Test HTTP status code

```
$ http localhost:8000/happy
HTTP/1.1 200 OK
content-length: 4
content-type: application/json
date: Sun, 05 Feb 2023 04:37:32 GMT
server: uvicorn

":)"
```

Headers

You can inject HTTP response headers, as in Example 3-30 and the test Example 3-31 (you don't need to return `response`.):

Example 3-30. Set HTTP headers

```
from fastapi import Response

@app.get("/header/{name}/{value}")
def header(name: str, value: str, response:Response):
    response.headers[name] = value
    return "normal body"
```

Example 3-31. Test response HTTP headers

```
$ http localhost:8000/header/marco/polo
HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Wed, 31 May 2023 17:47:38 GMT
marco: polo
server: uvicorn

"normal body"
```

Response Types

Some response types (import these classes from `fastapi.responses`) include:

- `JSONResponse`: the default
- `HTMLResponse`
- `PlainTextResponse`
- `RedirectResponse`
- `FileResponse`
- `StreamingResponse`

I’ll say more about the last three in chapter 19.

For other output formats (also known as *MIME types*), there’s also a generic `Response` class, which needs:

- `content`: String or bytes
- `media_type`: The string MIME type

- `status_code`: HTTP integer status code
- `headers`: A dict of strings

Type Conversion

The path function can return anything, and by default (using `JSONResponse`) FastAPI will convert it to a JSON string and return it, with the matching HTTP response headers `Content-Length` and `Content-Type`. This includes any Pydantic model class. But how does it do this? If you've used the Python `json` library, you've probably seen that it raises an exception when given some data types, such as `datetime`. FastAPI uses an internal function called `jsonable_encoder()` to convert any data structure to a "JSONable" Python data structure, then calls the usual `json.dumps()` to turn that into a JSON string. Example 3-32 shows a test:

Example 3-32. Use jsonable_encoder() to avoid JSON kabooms

```
import datetime
import pytest
from fastapi.encoders import jsonable_encoder
import json

@pytest.fixture
def data():
    return datetime.datetime.now()

def test_json_dump(data):
    with pytest.raises(Exception):
        json_out = json.dumps(data)

def test_encoder(data):
    out = jsonable_encoder(data)
    assert out
    json_out = json.dumps(out)
    assert json_out
```

Model Types and `response_model`

It's possible to have different classes with many of the same fields, except one is specialized for user input, one for output, and one for internal use:

- Remove some sensitive information from output (like *deidentifying* personal medical data, if you've encountered HIPPA requirements).

- Add fields to user input (like a creation date and time).

Example 3-X shows three related classes for a contrived case:

- `TagIn` is the class that defined what the user needs to provide (in this case, just a string called `tag`).
- `Tag` is made from a `TagIn`, and adds two fields: `created` (when this `Tag` was created) and `secret` (some internal string, maybe stored in a database, but never supposed to be exposed to the world).
- `TagOut` is the class that defines what can be returned to a user (by some lookup or search endpooint). It contains the `tag` field from the original `TagIn` object and its derived `Tag` object, plus the `created` field generated for `Tag`, but not `secret`.

Example 3-33. Model variations (model/tag.py)

```
from pydantic import BaseClass

class TagIn(BaseClass):
    tag: str

class Tag(BaseClass):
    tag: str
    created: datetime
    secret: str

class TagOut(BaseClass):
    tag: str
    created: datetime
```

You can return other data types than the default JSON from a FastAPI path function in different ways. One method is to use the `response_model` argument in the path decorator to goose FastAPI to return something else. FastAPI will drop any fields that were in the object that you returned but are not in the object specified by `response_model`. In example 3-X, I'll pretend that we wrote a new service module called *service/tag.py* with the `create()` and `get()` functions, that give this web module something to call. Thos lower-stack details don't matter here. The important point is the `get_one()` path function at the bottom, and the `response_model=TagOut` in its path decorator. That automatically changes an internal `Tag` object to a sanitized `TagOut` object.

Example 3-34. Return a different response type with response_model (web/tag.py)

```
from model.tag import TagIn, Tag, TagOut
import service.tag as service

@app.post('/'):
def create(tag_in: TagIn) -> TagIn:
    tag: Tag = Tag(tag=tag_str, created=datetime.utcnow(),
        secret="shhhh")
    service.create(tag)
    return tag_in

@app.get('/{tag_str}', response_model=TagOut)
def get_one(tag)_str: str) -> TagOut:
    tag: Tag = service.get(tag_str)
    return tag
```

Even though we returned a `Tag`, `response_model` will convert it to a `TagOut`.

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):

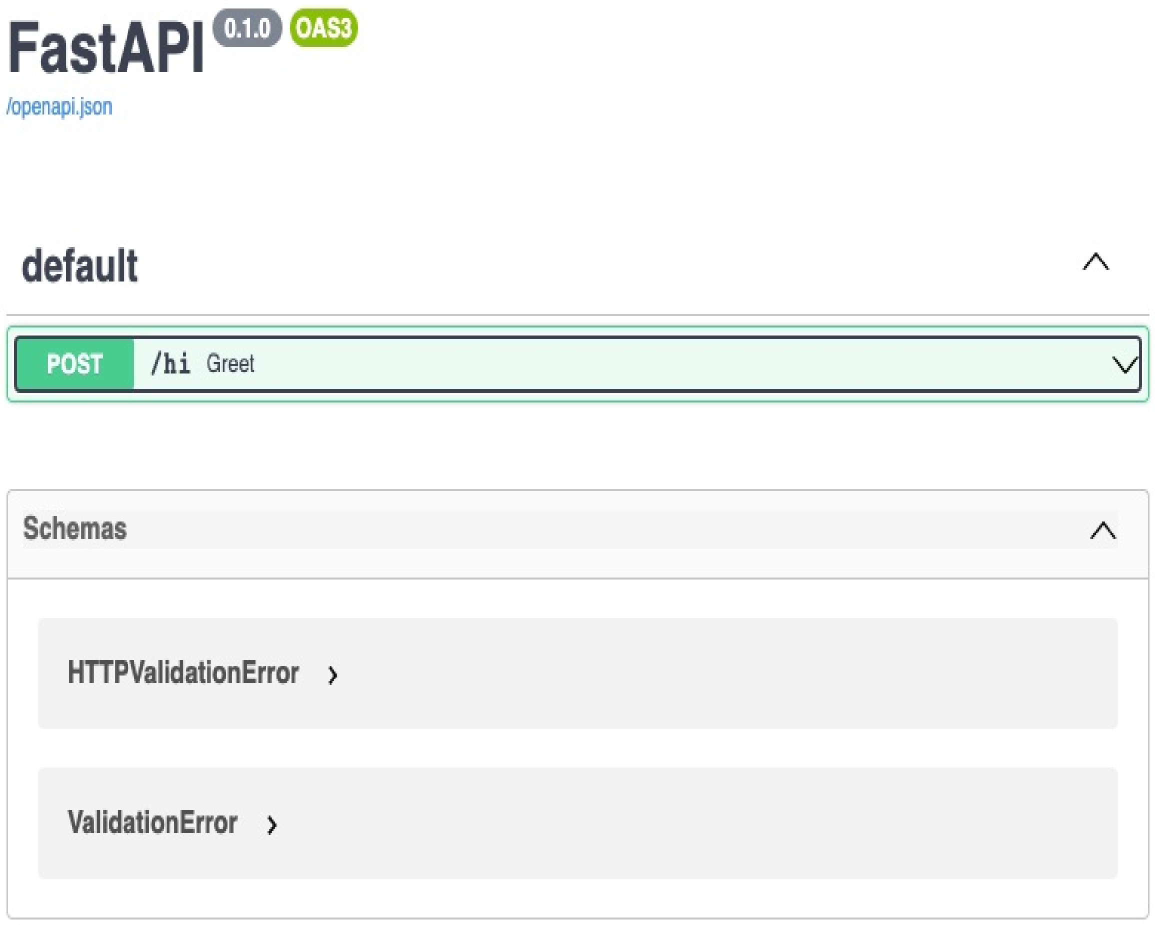


Figure 3-1. Generated documentation page

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:

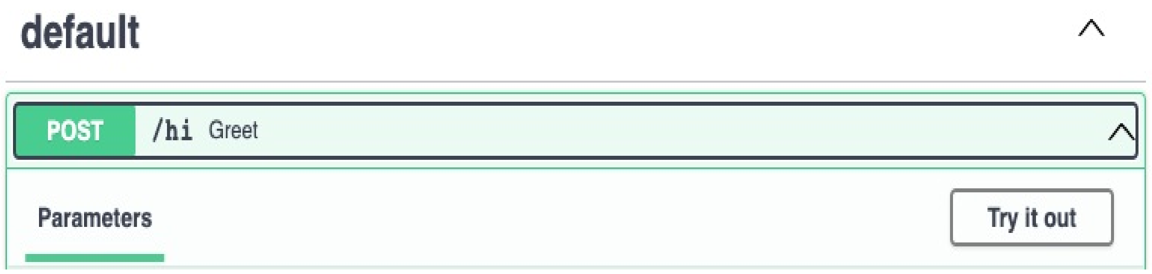
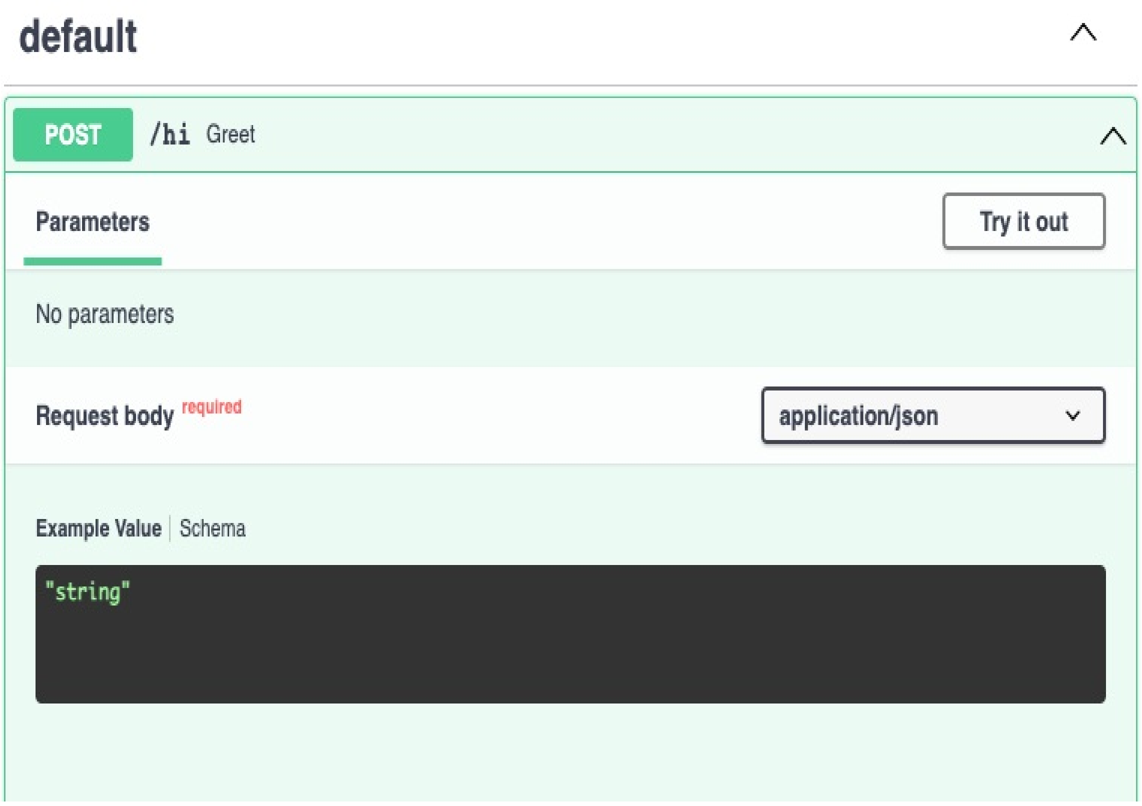


Figure 3-2. Open documentation page

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:

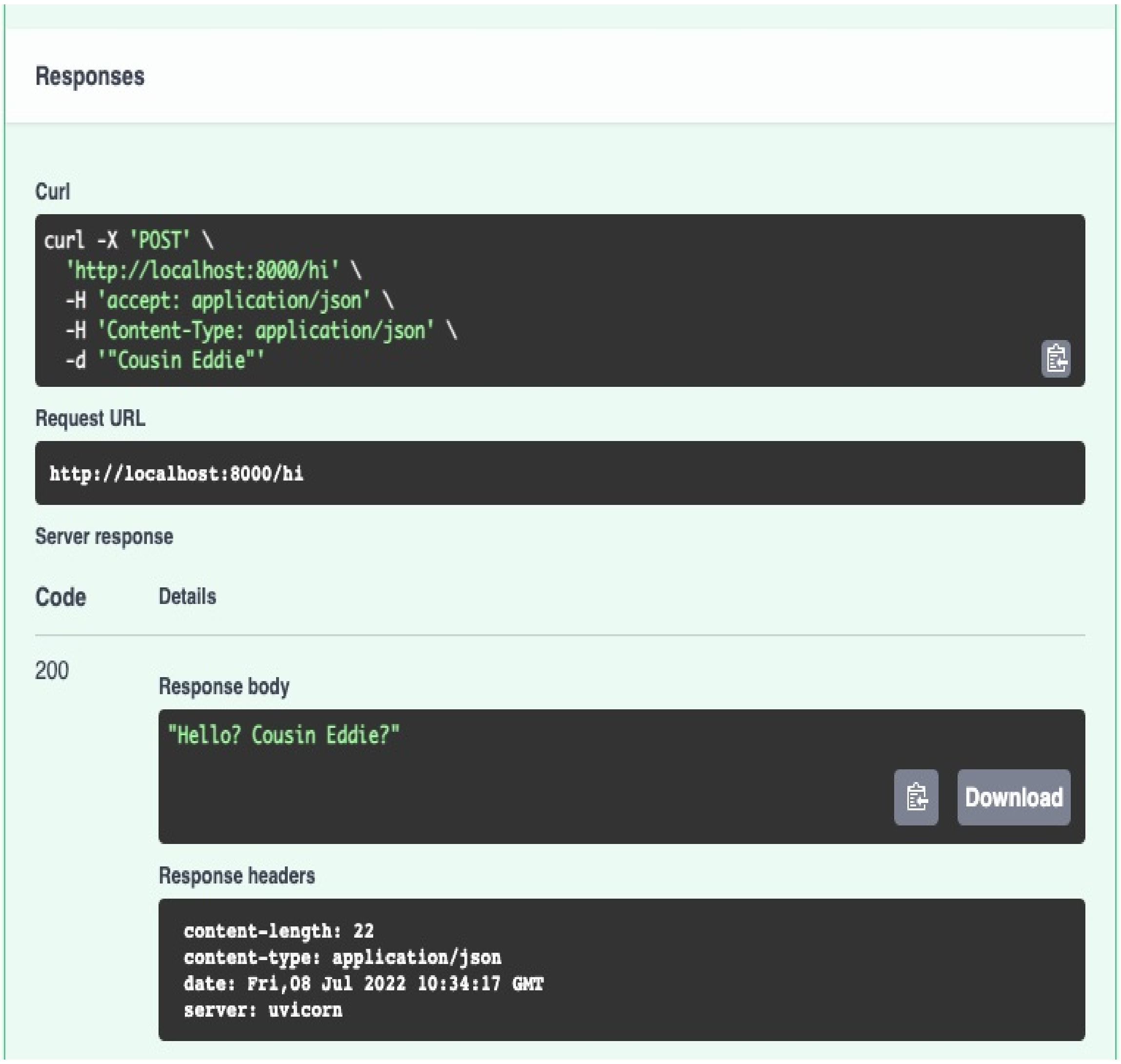


```
"string"
```

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:



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

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

200

```
"Hello? Cousin Eddie?"
```

```
content-length: 22
content-type: application/json
date: Fri,08 Jul 2022 10:34:17 GMT
server: uvicorn
```

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, FastAPI was used to create a website with a single endpoint. Different web clients tested it: a web browser, the httpie text program, the requests Python package, and the httpx Python package. Starting with a simple `GET` call, request arguments went to the server via the URL *path*, a query *parameter*, and an HTTP *header*. Then, the HTTP *body* was used to send data to a `POST` endpoint. Later, HTTP response utilities were shown. 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 8.

---

1 Watch your spell checker.

Chapter 4. Async, Concurrency, and Starlette Tour

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.

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 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 Simpsons episode, Homer crashes his car into a deer statue, followed by three lines of dialogue. Example 4-1 defines a normal Python function to `return` these lines as a list, and have the caller iterate over them:

Example 4-1. Use `return`

```
>>> 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.

Example 402 shows how a generator function would dole the lines out:

Example 4-2. Use `yield`

```
>>> 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. 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 Example 4-3:

Example 4-3. Dullness

```
>>> 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 Example 4-4 is a little different:

Example 4-4. Hilarity

```
>>> 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 4-5 below revisits the first FastAPI program from the previous chapter, and makes it asynchronous:

Example 4-5. 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 greet_async:app
```

The second, as in Example 4-6, is to call Uvicorn from inside the example code, when it's run as a main program instead of a module:

Example 4-6. 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("greet_async_uvicorn: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 Example 4-7:

Example 4-7. Using Starlette: `starlette_hello.py`

```
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 starlette_hello:app
```

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:
    a. Wait for the rug or wax to dry, but wave your bonus goodbye.
    b. 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 memeless 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. Pydantic, Type Hints, and Models Tour

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

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 (Example 5-1) and lists (Example 5-2) only let you access a member variable by its offset, so you have to remember what went where:

Example 5-1. Using a tuple

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

Example 5-2. Using a list

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

Example 5-3 shows that you can get a little more explanatory by defining names for the integer offsets:

Example 5-3. Using tuples and named 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 in Example 5-4, giving you access by descriptive keys:

Example 5-4. Using a dictionary

```
>>> 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.

In Example 5-5, a *named tuple* is a tuple that gives you access by integer offset *or* name:

Example 5-5.

```
>>> from collections import namedtuple
>>> CreatureNamedTuple = namedtuple("CreatureNamedTuple",
...     "name, description, location)
>>> 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, so the index needs to be an integer.

Example 5-6 defines a new Python `class`, and adds all the attributes with `self`. But you’ll need to do a lot of typing just to define them:

Example 5-6. Using a normal class

```
>>> class CreatureClass():
```

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

> **NOTE**
>
> You might think, what's so bad about that? With a regular class you can add more data (attributes), but especially behavior (methods). You might decide, one madcap day, to add a method that looks up an explorer's favorite songs. (This wouldn't apply to a creature[2].) But the use case here is just to move a clump of data undisturbed among the layers, and to validate on the way in and out. Also, methods are square pegs that would struggle to fit in the round holes of a database.

Does Python have anything similar to what other computer languages call a *record* or a *struct* (a group of names and values)? A recent addition to Python is the *dataclass*. Example 5-7 shows how all that `self` stuff disappears with dataclasses:

Example 5-7. Using a dataclass

```
>>> from dataclasses import dataclass
>>>
>>> @dataclass
... class CreatureDataClass():
...     name: str
...     description: str
...     location: 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

Alternatives

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 matching a pattern?

There are least three solutions that address at least some of these requirements:

- Dataclasses: Part of standard Python.
- Attrs: Third party, but a superset of dataclasses.
- Pydantic: Also third party, but integrated into FastAPI, so an easy choice if you’re already using FastAPI. And if you’re reading this book, that’s likely.

There’s a handy Youtube comparison of the three. One takeaway is that Pydantic stands out for validation, and its integration with FastAPI catches many potential data errors. Another is that Pydantic relies on inheritance (from the `BaseModel` class), and the other two use Python decorators to define their objects. This is more a matter of style.

In another comparison, Pydantic outperformed older validation packages like marshmallow and the intriguingly-named voluptuous. Another big plus for Pydantic is that it uses standard Python type hint syntax; older libraries predated type hints and rolled their own.

So I’m going with Pydantic in this book, but you may find uses for either of the alternatives if you’re not using FastAPI.

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
- Serialization and deserialization

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 in Example 5-8:

Example 5-8. Define a creature model: model.py

```
from pydantic import BaseModel
```

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

thing = Creature(name="yeti",
    description="Abominable Snowman",
    location="Himalayas")
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.

In Example 5-9, pass the arguments in any order if you include their names:

Example 5-9. Create a Creature

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

For now, Example 5-10 defines a teeny source of data; in later chapters, databases will do this. The type hint `list[Creature]` tells Python that this is a list of `Creature` objects only.

Example 5-10. Define fake data in data.py

```
from model import Creature

_creatures: list[Creature] = [
    Creature(name="yeti",
             description="Abominable Snowman",
             location="Himalayas"),
    Creature(name="sasquatch",
             description="Bigfoot",
             location="North America")
]
```

```
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.

Example 5-11 lists *web.py*, a file that defines a FastAPI web endpoint:

Example 5-11. Define a FastAPI web endpoint: 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 in Example 5-12:

Example 5-12. Start Uvicorn

```
$ 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, Example 5-13 accesses it with the Httpie web client (try your browser or the Requests module if you like, too):

Example 5-13. Test with Httpie

```
$ 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",
        "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
```

Validate Types

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 reply on any web code; it's a data thing).

Example 5-14 lists *test1.py*:

Example 5-14. Test the Creature model

```
from model import Creature

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

Now try it in Example 5-15:

Example 5-15. Run the test

```
$ python test1.py
Traceback (most recent call last):
  File ".../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.

Validate 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 nunber of elements

These are specified in the type parts of the model.

Example 5-1 ensures that the `name` field is always at least two characters long. Otherwise, `""` (an empty string) is a valid string.

Example 5-16. See a validation failure

```
>>> from pydantic import BaseModel, constr
>>>
>>> class Creature(BaseModel):
...     name: constr(min_length=2)
...     description: str
...     location: str
...
>>> bad_creature = Creature(name="!",
...     description="it's a raccoon",
...     location="your attic")
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*. Example 5-17 uses an alternative, the Pydantic `Field` specification:

Example 5-17. Another validation failure, using Field

```
>>> from pydantic import BaseModel, Field
>>>
>>> class Creature(BaseModel):
...     name: str = Field(..., min_length=2)
...     description: str
...     location: 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.

2 Except that small group of yodeling yetis (a good name for a band).

Chapter 6. Dependencies

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.

Example 6-1 shows a `user_dep()` dependency function that takes name and password string arguments, and just returns `True` if the user is valid. For tgis first version, let's have it return `True` for anything.

Example 6-1. A dependency function

```
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.

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`

Eexample 6-2 uses this to return a fixed user `name` and a `valid` boolean:

Example 6-2. Return a 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 in Example 6-3:

Example 6-3. Define 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 Paths

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. Example 6-4 sketches the idea:

Example 6-4. Define a subrouter dependency

```
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 you top-level FastAPI application object, you can add depenencies to it that will apply to all of its path functions, as shown in Example 6-5:

Example 6-5. Define app-level dependencies

```
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. Framework Comparisons

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

—Klaus Kinski

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

At the top level, Flask and FastAPI both use a decorator to associate a route with a web endpoint. In Example 7-1, 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.

Example 7-2 shows 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

In Example 7-3, 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 in Example 7-4 is:

Example 7-4. Flask query parameter example

```
from flask import Flask, request, jsonify

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

In Example 7-5, 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 Example 7-6:

Example 7-6. Flask body example

```
from flask import Flask, request, jsonify

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

Finally, let’s repeat example 3.11 in Example 7-7:

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:

Example 7-8. Flask header example

```
from flask import Flask, request, jsonify

app = Flask(&lowbar;&lowbar;name&lowbar;&lowbar;)
```

```
@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.

Unline 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, especiallyy 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: let’s build a website already.

Part III. Making a Website

Part II was a quick tour of FastAPI, to get you up to speed quickly. This part will go wider and deeper into the details.

We’ll build a medium sized web service to access and manage data about cryptids — imaginary creatures — and the equally fictitious explorers who seek them.

The full service will have three layers, as I’ve discussed earlier:

- *Web*: the web interface
- *Service*: the business logic
- *Data*: the precious DNA of the whole thing

Plus these cross-layer components:

- *Model*: Pydantic data definitions
- *Test*: unit, integration, and end-to-end tests

The site design will address:

- What belongs inside each layer?
- What is passed between layers?
- Can we change/add/delete things later without breaking anything?
- If something breaks, how do I find and fix it?
- What about security?
- Can it scale and perform well?
- Can we keep all of this as clear and simple as possible?

- Why do I ask so many questions? Why, oh why?

Chapter 8. Web Layer

Preview

Chapter 3 was a quick look at how to define FastAPI web endpoints, pass simple string inputs to them, and get responses. This chapter goes further into the top layer of a FastAPI application — which could also be called an *interface* or *router* layer — and its integration with the service and data layers.

As before, I'll start with small examples. Then I'll introduce some structure, dividing layers into subsections to allow for cleaner development and growth. The less code we write, the less we'll need to remember and fix later.

The basic sample data in this book concern imaginary creatures, or *cryptids*, and their explorers. You may find parallels with other domains of information.

What do we do with information, in general? Like most websites, we'll provide ways to:

- Retrieve
- Create
- Modify

- Replace
- Delete

Starting from the top, we’ll create web endpoints that perform these functions on our data. At first, we’ll provide fake data to make them work with any web client. In the following chapters, we’ll move the fake data code down into the lower layers. At each step, we’ll ensure that the site still works and passes data through correctly. Finally, in chapter 11, we’ll drop the faking and store real data in real databases, for a full end-to-end (web → service → data) website.

NOTE

Allowing any anonymous visitor to perfoem all of these actions will be an object lesson in “why we can’t have nice things”. The next chapter discusses the “auth” (authentication and authorization) needed to define roles and limit who can do what. For the rest of the current chapter, we’ll sidestep auth and just show how to handle the raw web functions.

Interlude: Top-Down, Bottom-Up, Middle-Out?

When designing a web site, you could start from:

- The web layer and work down.
- The data layer and work up.
- The service layer and work out in both directions.

Do you already have a database, installed and loaded with data, just pining for a way to share it with the world? If so, you may want to tackle the data layer’s code and tests first, then the service layer, and write the web layer last.

If you’re following domain-driven design, you might start in the middle service layer, definining your core entities and data models.

Or you may want to evolve the web interface first, and fake calls to the lower layers until you know what you’ll expect of them.

You’ll find very good design discussions and recommendations in:

- Clean Architectures in Python
- Architecture Patterns with Python
- Microservice APIs

In these and other sources, you’ll see terms like *hexagonal architecture*, *ports*, and *adaptors*. Your choices on how to proceed largely depend on what data you have already, and how you want to approach the work of building a site.

I’m guessing that many readers of this book are mainly interested in trying out FastAPI and its related technologies, and don’t necessarily have a predefined mature data domain that they want to instrument right away.

So, in this book I’m taking the web-first approach — step by step, starting with essential parts, and adding others as needed on the way down. Sometimes experiments work, sometimes not. I’ll avoid the urge to stuff everything into this web layer at first.

NOTE

This web layer is just one way of passing data between a user and a service. There are alternate ways, such as by a CLI (command line) or SDK (software development kit). In other frameworks, you might see this web layer called a *view* or *presentation* layer.

RESTful API Design

HTTP is a way to get commands and data between web clients and servers. But, just as you can combine ingredients from your refrigerator in ways from ghastly to gourmet, some recipes for HTTP work better than others.

In chapter 1, I mentioned that *RESTful* became a useful, though sometimes fuzzy, model for HTTP development. RESTful designs have some core components:

- *Resources*: The things your application manages.
- *IDs*: Unique resource identifiers.
- *URLs*: Structured resource and id strings.

- *Verbs* or *actions*: Accompany URLs for different purposes:
    - `GET`: Retrieve a resource.
    - `POST`: Create a new resource.
    - `PUT`: Completely replace a resource.
    - `PATCH`: Partially replace a resource.
    - `DELETE`: Resource goes kaboom.

NOTE

You'll see disagreement about the relative merits of `PUT` versus `PATCH`. If you don't need to distinguish between a partial modification and a full one (replacement), then you may not need both.

General RESTful rules for combining verbs and URLS containing resources and ids use these patterns of path parameters (things between the `/` in the URL):

- *verb /resource/*: Apply *verb* to all resources of type *resource*.
- *verb /resource/id*: Apply *verb* to the *resource* with id *id*.

Using the example data for this book, a `GET` request to the endpoint `/thing` would return data on all explorers, but a `GET` request for `/thing/abc` would only give you data for the `thing` resource with id `abc`.

Finally, web requests often carry more information:

- sort results
- paginate results
- perform some other function

Parameters for these can sometimes be expressed as *path* parameters (tacked onto the end, after another `/`), but are often included as *query* parameters (*var=val* stuff after the `?` in the URL). Because URLs have size limits, large requests are often conveyed in the HTTP *body*.

NOTE

Most authors recommend using plurals when naming the resource, and related things like API sections and database tables. I followed this advice for a long time, but now feel that singular names are simpler, for many reasons — some because of oddities of the English language — so your case may differ:

- Some words are their own plurals: `series`, `fish`
- Some words have irregular plurals: `children`, `people`
- You need bespoke singular to/from plural conversion code in many places

For these reasons, I'm using a singular naming scheme in many places in this book. This is against usual RESTful advice, so feel free to ignore this if you disagree.

File and Directory Site Layout

Our data mainly concern creatures and explorers. Initially, we could define all the URLs and their FastAPI path functions for accessing their data in a single Python file. Let's resist that temptation, and start as though we were already a rising star in the cryptid web space. With a good foundation, cool new things are much easier to add.

First, pick a directory on your machine. Name it *fastapi*, or anything that will help you remember where you'll be messing with the code from this book. Within it, create the following subdirectories:

- *src*: Contains all the website code.
    - *web*: The FastAPI web layer.
    - *service*: The business logic layer.
    - *data*: The storage interface layer.
    - *model*: Pydantic model definitions.
    - *fake*: Provides early *stub* data.

Each of these directories will soon gain three files:

- *__init__.py*: Needed to treat this directory as a package.

- *creature.py*: Creature code for this layer.
- *explorer.py*: Explorer code for this layer.

There are *many* opinions on how to lay out sites for development. This design is intended to show the layer separation and leave room for future additions.

Some explanations are needed right now:

Those *__init__.py* files are empty. They're sort of a Python hack so their directory should be treated as a Python *package* that may be imported from.

The *fake* directory provides some hard-wired (*stub*) data to higher layers as the lower ones are built

Python's *import* logic doesn't work strictly with directory hierarchies. It relies on Python *packages* and *modules*. The *.py* files listed in the tree above are Python modules (source files). Their parent directories are packages *if* they contain an *__init__.py* file. (This is sort of a hack, to help tell Python that, if you have a directory called *sys* and you type `import sys`, whether you actually want the system one or your local one.)

Python programs can import packages and modules. The Python interpreter has a builtin `sys.path` variable. which includes the location of the standard Python code. The environment variable `PYTHONPATH` is an empty or colon-separated string of directory names that tells Python what parent directories to check before `sys.path` to find the imported modules or packages. So, if you change to your new *fastapi* directory, type this (on Linux or macOS) to ensure that the new code under it will be checked first when importing:

```
$ export PYTHONPATH=$PWD/src
```

(That `$PWD` means Print Working Directory, and saves you from typing the full path to your *fastapi* directory, altjough you can if you want. And the `src` part means to look only in there for modules and packages to import.)

To set the `PWD` environment variable under Windows, see this link.

Whew.

The first website code

The next few sections discuss how to use FastAPI to write requests and responses for a RESTful API site. Then, I'll begin to apply these to our actual, increasingly gnarly, site.

Let's begin with Example 8-1. Within *src*, make this new top-level *main.py* program that will start the Uvicorn program and FastAPI package:

Example 8-1. The main program, main.py

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def top():
    return "top here"

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", reload=True)
```

That `app` is the FastAPI object that ties everything together. Uvicorn's first argument is `"main:app"` because the file is called *main.py*, and the second is `app`, the name of the FastAPI object.

Uvicorn will keep on running, and restart if any code changes in the same directory or any subdirectories. Without `reload=True`, each time you modify your code, you'd need to kill and restart Uvicorn manually. In many of the following examples, I just keep changing the same *main.py* file and forcing a restart, instead of creating *main2.py*, *main3.py*, and so on.

Fire up *main.py* in Example 8-2:

Example 8-2. Run the main program

```
$ python main.py &
INFO:     Will watch for changes in these directories:
[/Users/williamlubanovic/fastapi]
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to
quit)
INFO:     Started reloader process [92543] using StatReload
INFO:     Started server process [92551]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

That final & puts the program into the background, and you can run other programs in the same terminal window if you like. Or omit the & and run your other code in a different window or tab.

Now you can access the site localhost:8000 with a browser or any of the test programs that you've seen so far. Example 8-3 uses Httpie:

Example 8-3. Test the main program

```
$ http localhost:8000
HTTP/1.1 200 OK
content-length: 8
content-type: application/json
date: Sun, 05 Feb 2023 03:54:29 GMT
server: uvicorn

"top here"
```

From now on, as you make changes, the web server should restart automatically. If there's some error that kills it, restart it with `python main.py` again.

Example 8-4 adds another test endpoint, using a *path* parameter (part of the URL):

Example 8-4. Add an endpoint

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def top():
    return "top here"

@app.get("/echo/{thing}")
def echo(thing):
    return f"echoing {thing}"

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

As soon as you save your changes to *main.py* in your editor, the window where your web server is running should print something like this:

```
WARNING:  StatReload detected changes in 'main.py'. Reloading...
INFO:     Shutting down
```

```
  INFO:     Waiting for application shutdown.
  INFO:     Application shutdown complete.
  INFO:     Finished server process [92862]
  INFO:     Started server process [92872]
  INFO:     Waiting for application startup.
  INFO:     Application startup complete.
```

Example 8-5 shows if the new endpoint was handled correctly (the `-b` only prints the response body):

Example 8-5. Test new endpoint

```
$ http -b localhost:8000/echo/argh
"echoing argh"
```

In the rest of this chapter, we'll add more endpoints to *main.py*.

Requests

An HTTP request consists of a text *header* followed by one or more *body* sections. You could write your own code to parse HTTP into Python data structures, but you wouldn't be the first. In your web application, it's more productive to have these details done for you by a framework.

FastAPI's dependency injection is particularly useful here. Data may come from different parts of the HTTP message, and you've already seen how you can specify one or more of these dependencies to say where:

- `Header`: In the HTTP headers
- `Path`: In the URL
- `Query`: After the `?` in the URL
- `Body`: In the HTTP body

Other, more indirect, sources include:

- Environment variables
- Configuration settings

These are discussed in chapter 13.

Example 8-6 features an HTTP request, using our old friend Httpie, and ignoring the returned HTML body data:

Example 8-6. HTTP request and response headers

```
$ http -p HBh http://example.com/
GET / HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: example.com
User-Agent: HTTPie/3.2.1



HTTP/1.1 200 OK
Age: 374045
Cache-Control: max-age=604800
Content-Encoding: gzip
Content-Length: 648
Content-Type: text/html; charset=UTF-8
Date: Sat, 04 Feb 2023 01:00:21 GMT
Etag: "3147526947+gzip"
Expires: Sat, 11 Feb 2023 01:00:21 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (cha/80E2)
Vary: Accept-Encoding
X-Cache: HIT
```

In the first, it asked for the top page at *example.com* (a free website that anyone can use in, well, examples). It only asked for a URL, with no parameters anywhere else. The first block of lines is the HTTP request headers sent to the website, and the next block contains the HTTP response headers.

NOTE

Most test examples from here on won’t need all those request and response headers, so you’ll see more use of `http -b`.

Multiple Routers

Most web services handle multiple kinds of resources. Although you could throw all your path handling code in a single file and head off to happy hour somewhere, it’s often handy to use multiple *subrouters* instead of the single `app` variable that most of the examples up to now have used.

Under the *web* directory (in the same directory as the *main.py* file that you’ve been modifying so far), make a file called *explorer.py*, as in Example 8-7:

Example 8-7. APIRouter use in web/explorer.py

```
from fastapi import APIRouter

router = APIRouter(prefix = "/explorer")

@router.get("/")
def top():
    return "top explorer endpoint"
```

Now, Example 8-8 gets the top-level application *main.py* to know that there’s a new subrouter in town, which will handle all URLs that start with `/explorer`:

Example 8-8. Connect the main application (main.py) to the subrouter

```
from fastapi import FastApi
from .web import explorer

app = FastAPI()

app.include_router(explorer.router)
```

This new file will be picked up by Uvicorn. As usual, test in Example 8-9 instead of assuming it will work:

Example 8-9. Test new subrouter

```
$ http -b localhost:8000/explorer/
"top explorer endpoint"
```

Build the Web Layer

Now let’s start adding the actual core functions to the web layer. Initially, fake all the data in the web functions themselves. The next chapter, will move the fake data stuff to corresponding service functions, and in the data layer chapter, to the data functions. Finally, an actual database will be added for the data layer to access. At each development step, calls to the web endpoints should still work.

Define Data Models

First, define the data that we'll be passing among levels. Our *domain* contains explorers and creatures, so let's define minimal initial Pydantic models for them. Other ideas might come up later, like expeditions, journals, or ecommerce sales of coffee mugs. But for now, just include the two breathing (usually, in the case of creatures) models in Example 8-10:

Example 8-10. Model definition in model/explorer.py

```python
from pydantic import BaseModel

class Explorer(BaseModel):
    name: str
    nationality: str
```

Example 8-11 resurrects the `Creature` definition from earlier chapters:

Example 8-11. Model definition in model/creature.py

```python
from pydantic import BaseModel

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

These are very simple initial models. I didn't use any of Pydantic's features, such as required vs. optional, or constrained values. This simple code can be enhanced later without massive logic upheavals.

Stub and Fake Data

Also known as *mock data*, *stubs* are canned results that are returned without calling the normal "live" modules. They're a quick way to test your routes and responses.

A *fake* is a standin for a real data source, that performs at least some of the same functions. An example is an in-memory class that mimics a database. I'll be making some fake data in this chapter and the next few, as I fill in the code that defines ths layers and their communication. In chapter 10, I'll define an actual live data store (a database) to replace these fakes.

Create Common Functions Through the Stack

Similar to the data examples, the approach to building this site is exploratory. Often it isn’t clear what will eventually be needed, so let’s start with some pieces that would be common to similar sites. Providing a front-end for data usually requires some ways to:

- *Get* one, some, all
- *Create*
- *Replace* completely
- *Modify* partially
- *Delete*

Essentially, these are the “CRUD” basics from databases, although I’ve split the “U” into partial (*modify*) and complete (*replace*) functions. Maybe this distinction will prove unnecessary! It depends on where the data lead.

Create Fake Data

Working top-down, I’ll duplicate some functions in all three levels. To save typing, in Example 8-12 I’ll introduce the top-level directory called *fake*, with modules providing fake data on explorers and creatures:

Example 8-12. New module fake/explorer.py:

```
from model.explorer import Explorer

# fake data, replaced in chapter 10 by a real database and SQL
_explorers = [
    Explorer(name="Claude Hande",
             nationality="France"),
    Explorer(first_name="Noah Weiser",
             nationality="Germany"),
    ]

def get_all() -> list[Explorer]:
    """Return all explorers"""
    return _explorers

def get_one(name: str) -> Explorer | None:
    for _explorer in _explorers:
        if _explorer.name == name:
            return _explorer
    return None

# The following are non-functional for now,
# so they just act like they work, without modifying
# the actual fake _explorers list:
def create(explorer: Explorer) -> Explorer:
    """Add an explorer"""
    return explorer

def modify(explorer: Explorer) -> Explorer:
    """Partially modify an explorer"""
    return explorer

def replace(explorer: Explorer) -> Explorer:
    """Completely replace an explorer"""
    return explorer

def delete(name: str) -> bool:
    """Delete an explorer; return None if it existed"""
    return None
```

The creature setup in Example 8-13 is very similar:

Example 8-13. New module fake/creature.py:

```
from model.creature import Creature

# fake data, until we use a real database and SQL
_creatures = [
    Creature(name="yeti",
             description="Abominable Snowman",
             location="Himalayas"),
    Creature(name="bigfoot",
             description="AKA Sasquatch, the New World "
                         "Cousin Eddie of the yeti",
             location="North America"),
    ]

def get_all() -> list[Creature]:
    """Return all creatures"""
    return _creatures

def get_one(name: str) -> Creature | None:
    """Return one creature"""
    for _creature in _creatures:
        if _creature.name == name:
            return _creature
    return None

# The following are non-functional for now,
# so they just act like they work, without modifying
# the actual fake _creatures list:
def create(creature: Creature) -> Creature:
    """Add a creature"""
    return creature

def modify(creature: Creature) -> Creature:
    """Partially modify a creature"""
    return creature

def replace(explorer: Creature) -> Creature:
    """Completely replace a creature"""
    return creature

def delete(name: str):
    """Delete a creature; return None if it existed"""
    return None
```

NOTE

Yes, the module functions are almost identical. They'll change later, when a real database arrives and must handle the differing fields of the two models.

Also, I've used separate functions here, rather than defining a `Fake` class or abstract class. A module has its own namespace, so it's an equivalent way of bundling data and functions.

Now let's modify the web functions in Examples 8-13 and 8-14. Preparing to build out the later layers (Service and Data), import the fake data provider that was just defined above, but name it `service` here: `import fake.explorer as service`. In chapter 9, I'll:

- Make a new *service/explorer.py* file.
- Import the fake data there.
- Make *web/explorer.py* import the new service module instead of the fake module.

In chapter 10, I'll do the same in the data layer. All of this is just adding parts and wiring them together, with as little code rework as possible. I don't turn on the electricity (i.e., a live database and persistent data) until later in chapter 10.

Example 8-14. New endpoints for web/explorer.py

```
from fastapi import APIRouter
from model.explorer import Explorer
import fake.explorer as service

router = APIRouter(prefix = "/explorer")

@router.get("/")
def get_all() -> list[Explorer]:
    return service.get_all()

@router.get("/{name}")
def get_one(name) -> Explorer | None:
    return service.get_one(name)

# all the remaining endpoints do nothing yet:
@router.post("/")
def create(explorer: Explorer) -> Explorer:
    return service.create(explorer)

@router.patch("/")
def modify(explorer: Explorer) -> Explorer:
    return service.modify(explorer)

@router.put("/")
def replace(explorer: Explorer) -> Explorer:
    return service.replace(explorer)

@router.delete("/{name}")
def delete(name: str):
    return None
```

And now, the same for `/creatures` endpoints. Yes, this is very similar cut and paste for now, but doing this up front simplifies changes later on — and there will always be changes later on.

Example 8-15. New endpoints for web/creature.py

```
from fastapi import APIRouter
from model.explorer import Explorer
import fake.creature as service

router = APIRouter(prefix = "/creature")

@router.get("/")
def get_all() -> list[Creature]:
    return service.get_all()
```

```
@router.get("/{name}")
def get_one(name) -> Creature:
    return service.get_one(name)

# all the remaining endpoints do nothing yet:
@router.post("/")
def create(explorer: Creature) -> Creature:
    return service.create(creature)

@router.patch("/")
def modify(creature: Creature) -> Creature:
    return service.modify(creature)

@router.put("/")
def replace(creature: Creature) -> Creature:
    return service.replace(creature)

@router.delete("/{name}")
def delete(name: str):
    return service.delete(name)
```

The last time I poked at *main.py*, it was to add the subrouter for `/explorer` URLs. Now, let's add another for `/creature` in Example 8-15:

Example 8-16. Add creature subrouter to main.py

```
import uvicorn
from fastapi import FastApi
from web import explorer, creature

app = FastAPI()

app.include_router(explorer.router)
app.include_router(creature.router)

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

Did all of that work? If you typed or pasted everything exactly, Uvicorn should have restarted the application. Let's try some manual tests.

Test!

Chapter 12 will show how the use of Pytest to automate testing at various levels. Example 8-16 to 8-20 do some manual Web-layer tests of the explorer endpoints with Httpie:

Example 8-17. Test the get all endpoint

```
$ http -b localhost:8000/explorer/
[
    {
        "nationality": "France",
        "name": "Claude Hande"
    },
    {
        "nationality": "Germany",
        "first_name": "Noah Weiser"
    }
]
```

Example 8-18. Test the get one endpoint

```
$ http -b localhost:8000/explorer/"Noah Weiser"
{
    "nationality": "Germany",
    "name": "Noah Weiser"
}
```

Example 8-19. Test the replace endpoint

```
$ http -b PUT localhost:8000/explorer/"Noah Weiser"
{
    "nationality": "Germany",
    "first_name": "Noah Weiser"
}
```

Example 8-20. Test the modify endpoint

```
$ http -b PATCH localhost:8000/explorer/"Noah Weiser"
{
    "nationality": "Germany",
    "first_name": "Noah Weiser"
}
```

Example 8-21. Test the delete endpoint

```
$ http -b DELETE localhost:8000/explorer/Noah%20Weiser
true

$ http -b DELETE localhost:8000/explorer/Edmund%20Hillary
false
```

You can do the same for the `/creature` endpoints.

Using the FastAPI Automated Test Forms

Besides the manual tests that I've used in most examples, FastAPI provides very nice automated test forms at the endpoints `/docs` and `/redocs`. They're two different styles for the same information, so I'll just show a little from the `/docs` pages here.

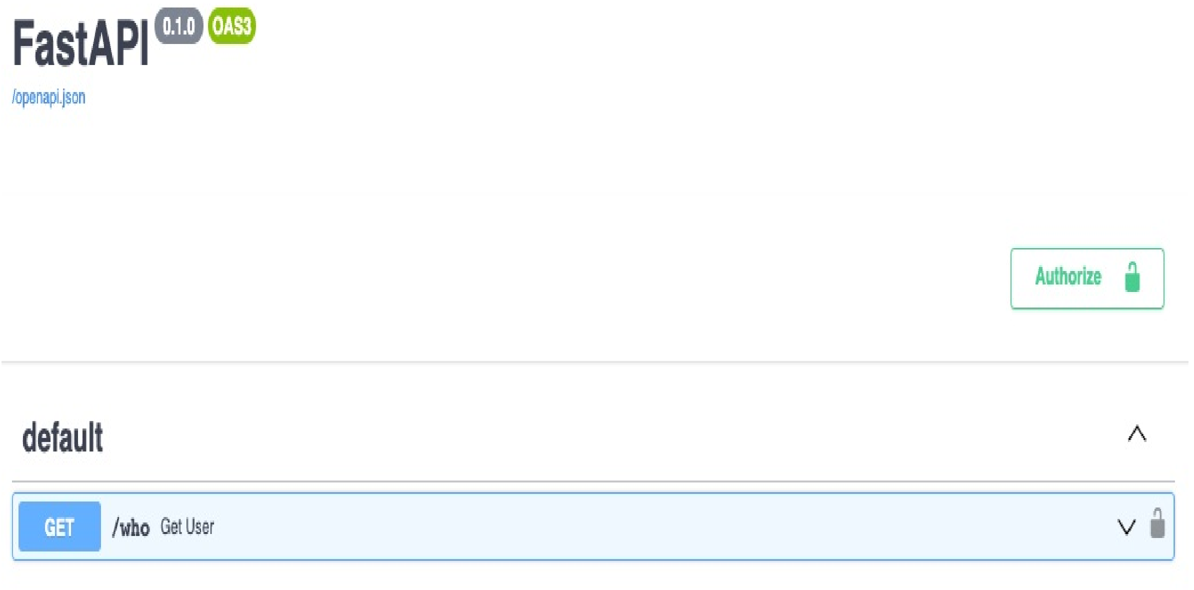
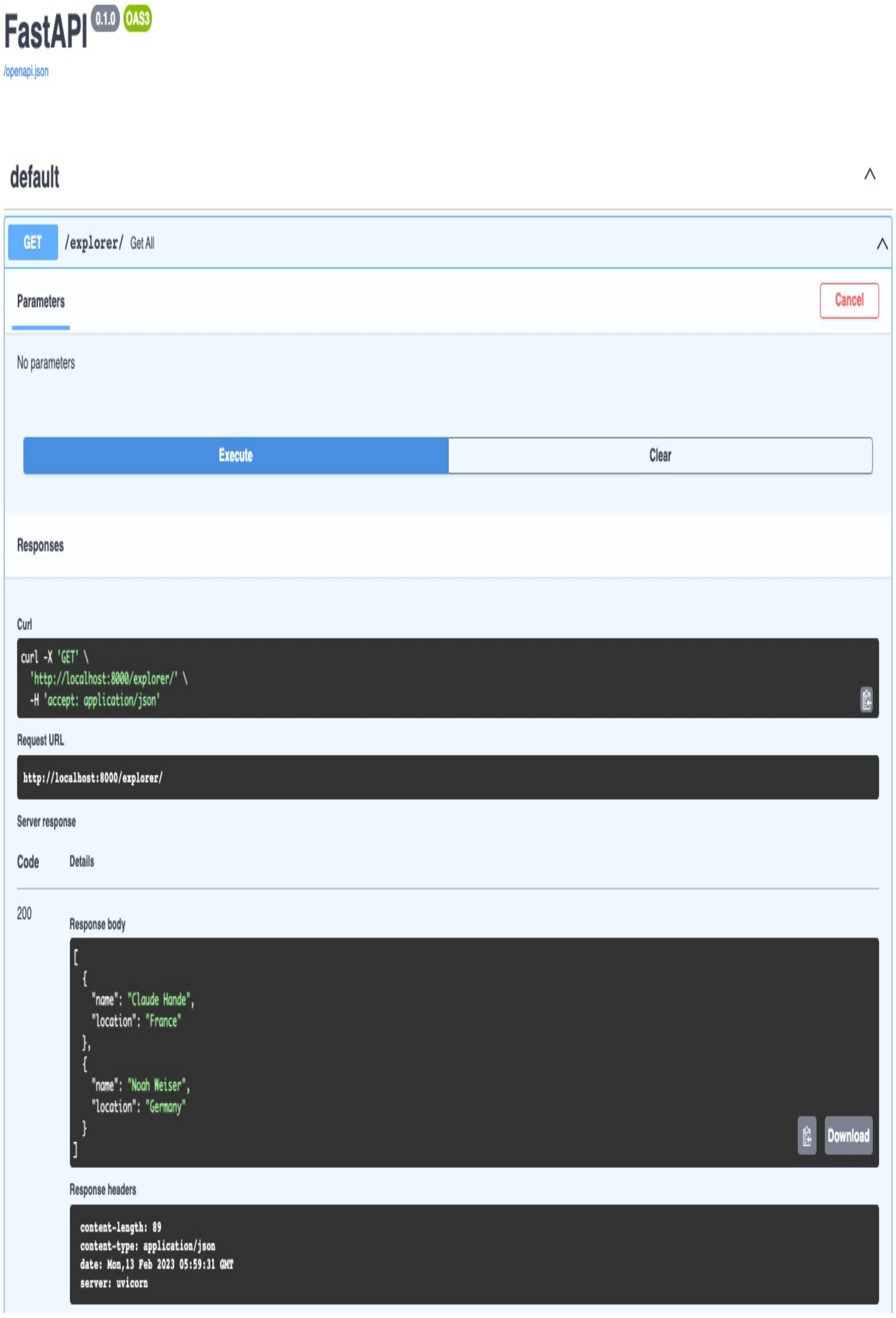


Figure 8-1. Generated documentation page

Try the first test:

- Click on the down-arrow to the right under the top `GET /explorer` section.
- That will open up a large light blue form.
- Click the blue `Execute` button on the left.
- You'll see the top section of the results in the following image.



```
curl -X 'GET' \
  'http://localhost:8000/explorer/' \
  -H 'accept: application/json'
```

```
http://localhost:8000/explorer/
```

```
[
  {
    "name": "Claude Hande",
    "location": "France"
  },
  {
    "name": "Noah Weiser",
    "location": "Germany"
  }
]
```

```
content-length: 89
content-type: application/json
date: Mon,13 Feb 2023 05:59:31 GMT
server: uvicorn
```

Figure 8-2. Generated results page for GET /explorer

In the lower `Response body` section, you’ll see the JSON returned for the (fake) explorer data that we’ve defined so far:

```
[
  {
    "name": "Claude Hande",
    "location": "France"
  },
  {
    "name": "Noah Weiser",
    "location": "Germany"
  }
]
```

Try all the others. For some (like `GET /explorer/{name}`), you’ll need to provide an input value. You’ll get a response for each, even though a few are still no-ops until the database code is added.

You can repeat these tests near the end of the upcomning service and data chapters, to ensure no data pipelines were punctured during these code changes.

Talking to the Service and Data Layers

Whenever a function in the web layer needs data that are managed by the data layer, it should ask the service layer to be an intermediary. This requires some more code, and may seem unnecessary, but it’s a good idea:

- As the label on the jar says, the Web layer deals with the web, and the Data layer deals with external data stores and services. It’s much safer to keep their respective details completely separate.
- The layers can be tested independently. Separation of layer mechanisms allows this.

NOTE

For a very small site, you could skip the service layer if it doesn’t really add any value. The next chapter initially defines service functions that do little more than pass requests and responses between the web and data layers. At least keep the web and data layers separate, though.

What does that Service layer function do? You’ll see in the next chapter. Hint: it talks to the Data layer, but in a hushed voice so the Web layer doesn’t know exactly what it’s saying. But it also defines any specific business logic, such as interactions between resources. Mainly, the Web and Data layers should not care what’s going on in there. (It’s a Secret Service.)

Pagination and Sorting

In web interfaces, when returning many or all things with URL patterns like `GET` */resource*, you often want to request the lookup and return of:

- Only one thing
- Possibly many things
- All things

How do you get our well-meaning but extremely literal-minded computer to do these things? Well, for the one thing case, the RESTful pattern that I mentioned earlier is to include the resource’s id in the URL path. When getting multiple resources, we may want to see the results in a particular order:

- *Sort*: Order all of the results, even if you only get a set of them at a time.
- *Paginate*: Return only some results at a time, respecting any sorting.

In each case, a group of user-specified parameters specifies wnat you want. It’s common to provide these as query parameters. Examples:

- *Sort*: `GET /explorer?sort=nationality`: Get all explorers, sorted by nationality.
- *Paginate*: `GET /explorer?offset=10&size=10`: Return (in this case, unsorted) explorers in places 10 through 19 of the whole list.
- *Both*: `GET /explorer?sort=nationality&offset=10&size=10`

Although you could specify these as individual query parameters, FastAPI's dependency injection can help:

- Define the sort and paginate parameters as a Pydantic model.
- Provide the parameters model to the `get_all()` path function with the `Depends` feature in the path function arguments.

Where should the sorting and pagination occur? At first, it seem simplest for the database queries to pass full results up to the Web layer, and use Python to carve up the data there. But that isn't very efficient. These tasks usually fit best in the Data layer, because databases are good at those things. I'll finally get around to some code for these in chapter 17, which has more database tidbits beyond those in chapter 10.

Review

This chapter filled out more details from chapter 3 and others. It began the process of making a full site for information on imaginary creatures and their explorers. Starting with the web layer, I defined endpoints with FastAPI path decorators and path functions. The path functions gather request data from wherever they live in the HTTP request bytes. Model data are automatically checked and validated by Pydantic. Path functions generally pass arguments to corresponding service functions, which are coming in the next chapter.

Chapter 9. Service Layer

What was that middle thing?

—Otto West, A Fish Called Wanda

This chapter expands on the Service layer — the middle thing.

Preview

A leaky roof can cost a lot of money. Leaky software isn’t as obvious, but can cost a lot of time and effort. How can you structure your application so that the layers don’t leak? In particular, what should and should not go into the Service layer in the middle?

Defining a Service

This layer is the heart of the website, its reason for being. It takes requests from multiple sources, accesses the data that are the DNA of the site, and returns responses.

Common service patterns include a combination of:

- Create / retrieve / change (partially or completely) / delete
- One thing / multiple things

At the RESTful router layer, the nouns are *resources*. In this book, our resources will initially include:

- Cryptids (imaginary creatures)
- People (cryptid explorers)

Later, it will be possible to define related resources like:

- Places
- Events (e.g., expeditions, sightings)

Layout

Here's the current file and directory layout:

- *main.py*
- *web*
  - *__init__.py*
  - *creature.py*
  - *explorer.py*
- *service*
  - *__init__.py*
  - *creature.py*
  - *explorer.py*
- *data*
  - *__init__.py*
  - *creature.py*

    - *explorer.py*
- *model*
    - *__init__.py*
    - *creature.py*
    - *explorer.py*
- *fake*
    - *__init__.py*
    - *creature.py*
    - *explorer.py*
- *test*

In this chapter, I’ll fiddle with files in the *service* directory.

Protection

One nice thing about layers is that you don’t have to worry about everything. The service layer only cares about what goes into and out of the data. As you’ll see in Chapter 11, a higher layer (in this book, *web*) can handle the auth messiness. The functions to create, modify, and delete should not be wide open, and even the `get` functions might eventually need some limits.

Functions

Let’s start with *creature.py*. At this point, the needs of *explorer.py* will be almost the same, and we can borrow almost everything. It’s so very tempting to write a single service file that handles both, but, almost inevitably, at some point we’ll need to handle them differently.

Also at this point, the service file is pretty much a passthrough layer. This is a case in which a little extra structure at the start will pay off later. Much as I did for *web/creature.py.py* and *web/explorer.py.py* in Chapter 8, I'll define service modules for both, and hook both of them up to their corresponding *fake* data modules for now (Examples 9-1 and 9-2).

Example 9-1. An initial service/creature.py file

```
from models.creature import Creature
import fake.creature as data

def get_all() -> list[Creature]:
    return data.get_all()

def get_one(name: str) -> Creature | None:
    return data.get(id)

def create(creature: Creature) -> Creature:
    return data.create(creature)

def replace(id, creature: Creature) -> Creature:
    return data.replace(id, creature)

def modify(id, creature: Creature) -> Creature:
    return data.modify(id, creature)

def delete(id, creature: Creature) -> bool:
    return data.delete(id)
```

Example 9-2. An initial service/explorer.py file

```
from models.explorer import Explorer
import fake.explorer as data

def get_all() -> list[Explorer]:
    return data.get_all()

def get_one(name: str) -> Explorer | None:
    return data.get(name)

def create(explorer: Explorer) -> Explorer:
    return data.create(creature)

def replace(id, explorer: Explorer) -> Explorer:
    return data.replace(id, creature)

def modify(id, explorer: Explorer) -> Explorer:
    return data.modify(id, creature)

def delete(id, explorer: Explorer) -> bool:
    return data.delete(id)
```

NOTE

The syntax of the `get_one()` function's return value (`Creature | None`) needs at least Python 3.9. For earlier versions, you need `Optional`:

```
from typing import Optional
...
def get_one(name: str) -> Optional[Creature]:
...
```

Test!

Now that the code base is filling out a bit, it's a good time to introduce automated tests. (The Web tests in the previous chapter have all been manual tests.) So let's make some directories:

- *test*: A top-level directory, alongside *web*, *service*, *data*, and *model*.
    - *unit*: Exercise single functions, but don't cross layer boundaries.
        - *web*: Web-layer unit tests.
        - *service*: Service-layer unit tests.
        - *data*: Data-layer unit tests.
    - *full*: Also known as *end-to-end* or *contract* tests, these span all layers at once. They address the API endpoints in the Web layer.

(The directories have the *test_* prefix or *_test* suffix for use by Pytest, which you'll start to see later in this chapter.)

Before testing, a few API design choices need to be made. What should be returned by the `get_one()` function if a matching `Creature` or `Explorer` isn't found? You can return `None`, as I did here. Or you could raise an exception. None of the built-in Python exception types deal directly with missing values:

- `TypeError` may be the closest, because `None` is a different type than `Creature`.

- `ValueError` is more suited for the wrong value for a given type, but I guess you could say that passing a missing string `id` to `get_one(id)` qualifies.
- You could define your own `MissingError` if you really want to.

Whichever method you choose, the effects will bubble up all the way to the top layer.

Let's go with the `None` alternative rather than the exception for now. After all, that's what "none" means, and deciding things is hard. Example 9-3 is a test:

Example 9-3. Service test test/unit/service/test_creature.py

```
from model.creature import Creature
from service import creature as code

sample = Creature(name="yeti",
        description="Abominable Snowman",
        location="Himalayas")

def test_create():
    resp = code.create(sample)
    assert resp == sample

def test_get_exists():
    resp = code.get_one("yeti")
    assert resp == sample

def test_get_missing():
    resp = code.get_one("boxturtle")
    assert data is None
```

Run the test in Example 9-4:

Example 9-4. Run service test

```
$ pytest -v test/unit/service/test_creature.py
test_creature.py::test_create PASSED                                   [ 16%]
test_creature.py::test_get_exists PASSED                               [ 50%]
test_creature.py::test_get_missing PASSED                              [ 66%]

======================== 3 passed in 0.06s =========================
```

NOTE

In Chapter 10, `get_one()` will no longer return `None` for a missing creature, and the `test_get_missing()` test above would fail. But that will be fixed.

Other Service-Level Stuff

We’re in the middle of the stack now — the part that really defines our site’s purpose. And so far, we’ve only used it to forward web requests to the (next chapter) data layer.

So far, this book has developed the site iteratively, building a minimal base for future work. As you learn more about what you have, what you can do, and what users might want, then you can branch out and experiment. Some ideas might only benefit larger sites, but technical site-helper ideas include:

- Logging
- Metrics
- Monitoring
- Tracing

The following sections discuss each of these. We’ll revisit them in Chapter 15 (Troubleshooting), to see if they can help to diagnose problems.

Logging

FastAPI logs each API call to an endpoint — including the timestamp, method, and URL — but not any data delivered via the body or headers.

Loki

(MORE)

Metrics, Monitoring, Observability

If you run a website, you probably want to know how it’s doing. For a API website, you might want to know what endpoints are being accessed, how many people are visiting, and so on. Statistics on such things are called *metrics*, and the gathering of them is *monitoring* or *observability*.

Pupular metrics tools nowadays include:

- Gather metrics: Prometheus
- Display metrics: Grafana:

(MORE)

Tracing

How well is your site performing? It’s common for things to be good overall, but with disappointing results here or there. Or the whole site may be a mess. Either way, it’s useful to have a tool that measures how long an API call takes, end to end. And not just overall time, but the time for each intermediate step. If something’s slow, you can find the weak link in the chain. This is *tracing*.

A new nonprofit called OpenTelemetry has taken earlier tracing products like Jaeger, and branded them as *OpenTelemetry*. It has a Python API, and at least one integration with FastAPI.

To install the Python package, `pip install opentelemetry-instrumentation-fastapi`.

(MORE)

Other

These will be discussed in chapter 13. Besides these, what about our domain — cryptids and anything associated with them? Besides bare details on explorers and creatures, what else might you want to take on? You may come up with new ideas that require changes to the models and other layers. Ideas might include:

- Link explorers to the creatures that they seek
- Sighting data
- Expeditions

- Photos and videos
- Sasquatch mugs and tshirts (Figure 9-1)

Figure 9-1. A word from our sponsor

Each of these will generally require one or more new models to be defined, and new modules and functions. Some of these will be added in Part IV, which is a gallery of applications added to the base built here in part III.

Review

In this chapter, I replicated some functions from the web layer, and moved the fake data that they worked with. This wasn’t to get paid by the line, but to initiate the new service layer. So far, it’s been a cookie cutter process, but it will evolve and diverge after this. The next chapter builds the final data layer, yielding a truly live website.

Chapter 10. Data Layer

If I'm not mistaken, I think Data was the comic relief on the show.

—Brent Spiner, Star Trek: The Next Generation

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 10th 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

This chapter finally creates a persistent home for our site's data, at last connecting the three layers. It uses the relational database SQLite, and introduces Python's database API, aptly named DB-API. Chapter 17 goes into much more detail on databases, including the SQLAlchemy package and non-relational databases.

DB-API

For over twenty years, Python has included a basic definition for a relational database interface called DB-API: PEP-249. Anyone who writes a Python driver for a relational database is expected to at least include support for DB-API, although other features may be included.

The main DB-API functions are these:

- Create a connection `conn` to the database with `connect()`.
- Create a cursor `curs` with `conn.cursor()`.
- Execute a SQL string `stmt` with `curs.execute(stmt)`.

The `execute...()` functions run a SQL statement *stmt* string with optional parameters (see below for the multiple ways in which parameters may be specified.).

- `execute(stmt)` if there are no parameters.
- `execute(stmt, params)`, with parameters *params* in a single sequence (list or tuple) or dict.
- `executemany(stmt, params_seq)`, with multiple parameter groups in the sequence *params_seq*.

There are *five* different ways of specifying parameters, and not all of them are supported by all database drivers. For a statement `stmt` that begins with `"select * from creature where"`, and we want to specify string parameters for the creature's `name` *or* `location`, the rest of the `stmt` string and its parameters would look like this:

Type	Statement part	Parameters part
qmark	`name=? or location=?`	`(name, location)`
numeric	`name=:0 or location=:1`	`(name, location)`
format	`name=%s or location=%s`	`(name, location)`
named	`name=:name or location=:location`	`{"name": name, "location": location}`
pyformat	`name=%(name)s or location=%(location)s`	`{"name": name, "location": location}`

The first three take a tuple argument, where the parameter order matches the `?`, `:N`, or `%s` in the statement. The last two take a dictionary, where the keys match the names in the statement.

So, the full call for the *named* style would look like Example 10-1:

Example 10-1. Example with named-style parameters.

```
stmt = """select * from creature where
    name=:name or location=:location"""
params = {"name": "yeti", "location": "Himalayas"}
curs.execute(stmt, params)
```

For SQL `INSERT`, `DELETE`, and `UPDATE` statements, the returned value from `execute()` tells you how it worked. For `SELECT`, you iterate over returned data row(s) (as Python tuples) with a `fetch` method:

- `fetchone()` return one tuple, or `None`.
- `fetchall()` returns a sequence of tuples.
- `fetchmany(num)` returns up to *num* tuples.

SQLite

Python includes support for one database (SQLite) with the module sqlite3 in its standard packages.

SQLite is unusual: there is no separate database server. All the code is in a library, and storage is in a single file. Other databases run separate servers, and clients communicate with them over TCP/IP, using specific protocols. Let’s use SQLite as the first physical data store for this website. Chapter 14 will include other databases, relational and not, as well as more advanced packages like SQLAlchemy and techniques like ORMs.

First, we need to define how the data structures we’ve been using in the website (*models*) can be represented in the database. So far, our only models have been simple and similar, but not identical: `Creature` and `Explorer`. They will change as we think of more things to do with them. and to let the data evolve without massive code changes.

Example 10-2 shows the bare DB-API code and SQL to create and work with the first tables. It uses *named* argument strings (where values are represented like `:name`), which are supported by the sqlite3 package.

Example 10-2. Create file data/creature.py using sqlite3

```
import sqlite3
from ..model.creature import Creature

DB_NAME = "cryptid.db"
conn = sqlite3.connect(DB_NAME)
curs = _conn.cursor()

def init():
    curs.execute("create table creature(name, description, location)")

def row_to_model(row: tuple) -> Creature:
    return Creature(name, description, location = row)

def model_to_dict(creature: Creature) -> dict:
    return creature.dict()

def get_one(name: str) -> Creature:
    qry = "select * from creature where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    row = curs.fetchone(res)
    return row_to_model(row)

def get_all(name: str) -> list[Creature]:
    qry = "select * from creature"
    curs.execute(qry)
    rows = list(curs.fetchall())
    return rows_to_models(rows)

def create(creature: Creature):
    qry = "insert into creature values"
          "(:name, :description, :location)"
    params = model_to_dict(creature)
    res = curs.execute(qry, params)

def modify(creature: Creature):
    return creature

def replace(creature: Creature):
    return creature

def delete(creature: Creature):
    qry = "delete from creature where name = :name"
    params = {"name": name}
    res = curs.execute(qry, params)
```

Near the top, the `init()` function makes the connection to sqlite3 and the database fake *cryptid.db*. It stores this in the variable `conn`; this is global within the `data/creature.py` module. Next, the `curs` variable is a *cursor* for iterating over data returned by executing a SQL `SELECT` statement; it's also global to the module.

Two utility functions translate between Pydantic models and DB-API:

- `row_to_model()` converts a tuple returned by a *fetch* function to a model object.
- `model_to_dict()` translates a Pydantic model to a dictionary, suitable for use as a *named* query parameter.

The fake CRUD functions that have been present so far in each layer down (Web → Service → Data) will now be replaced. They use only plain SQL and the DB-API methods in `sqlite3`.

These functions are very basic so far. They don't include support for filtering, sorting, or pagination — that's all coming in chapter 14.

Layout

So far, (fake) data have been modified in steps:

- *Chapter 8*: Made the fake `_creatures` list in *web/creature.py*.
- *Chapter 8*: Made the fake `_explorers` list in *web/explorer.py*.
- *Chapter 9*: Moved fake `_creatures` to *service/creature.py*.
- *Chapter 9*: Moved fake `_explorers` to *service/explorer.py*.

Now the data have moved for the last time, down to *data/creature.py*. But they're not fake anymore: they're real live data, persisting in the SQLite database file *cryptids.db*. Creature data, again by lack of imagination, are stored in the SQL table `creature` in this database.

Once you save this new file, Uvicorn should restart from your top *main.py*, which calls *web/creature.py*, which calls *service/creature.py*, and finally down to this new *data/creature.py*.

Making It Work

There's one small problem: this module never calls its `init()` function, so there's no SQLite `conn` or `curs` for the other functions to use.

This a configuration issue: how to provide the database information at startup time. Possibilities include:

- Hard-wiring the database info in the code, as in the code above.
- Passing the info down through the layers. But this would violate the separation of layers; the Web and Service layers should not know the internals of the Data layer.
- Passing the info from a different external source, such as:
  - A *config file*.
  - An *environment variable*.

The environment variable is simple, and endorsed by recommendations like the Twelve Factor App. The code can include a default value if the environment variable isn't specified. This approach can also be used in testing, to provide a separate test database from the production one.

In Example 10-3, let's define an environment variable called `CRYPTID_SQLITE_DB`, with the default value `cryptid.db`. Make a new file called *data/init.py* for the new database initialization code so it can also be reused for the explorer code.

Example 10-3. New data initialization module data/init.py

```python
import sqlite3
import os

_dbname = os.environ.get("CRYPTID_SQLITE_DB", "cryptid.db")
conn = sqlite3.connect(db_name)
curs = conn.cursor()
```

A Python module is a *singleton*, only called once despite multiple imports. So, the initialization code in *init.py* is only run once, when the first import of it occurs.

Last, modify *data/creature.py* in Example 10-4 to use this new module instead:

- Mainly, drop lines 4 through 8.
- Oh, and create the `creature` table in the first place!
- The table fields are all SQL `text` strings. This is the default column type in SQLite (unlike most SQL databases), so I didn't need to include `text` earlier, but being explicit doesn't hurt.
- The `if not exists` avoids clobbering the table after it's been created.
- The `name` field is the explicit `primary key` for this table. If this table ever houses lots of explorer data, that key will be necessary for fast lookups. The alternative is the dreaded *table scan*, where the database code needs to look at every row until it finds a match for `name`.

Example 10-4. Add database configuration to data/creature.py

```
from .init import conn, curs
from ..model.creature import Creature

curs.execute("""create table if not exists creature(
                name text primary key,
                description text,
                location text)""")

def row_to_model(row: tuple) -> Creature:
    return Creature(name, description, location = row)

def model_to_dict(creature: Creature) -> dict:
    return creature.dict()

def get_one(name: str) -> Creature:
    qry = "select * from creature where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    return row_to_model(curs.fetchone())

def get_all() -> list[Creature]:
    qry = "select * from creature"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(creature: Creature) -> Creature:
    qry = "insert into creature values"
          "(:name, :description, :location)"
    params = model_to_dict(creature)
    curs.execute(qry, params)
    return get_one(creature.name)

def modify(creature: Creature) -> Creature:
    qry = """update creature
             set location=:location,
                 name=:name,
                 description=:description
             where name=:name0"""
    params = model_to_dict(creature)
    params["name0"] = creature.name
    res = curs.execute(qry, params)
    return get_one(creature.name)

def delete(creature: Creature) -> bool:
    qry = "delete from creature where name = :name"
    params = {"name": name}
    res = curs.execute(qry, params)
    return bool(res)
```

By importing `conn` and `curs` from *init.py*, it's no longer necessary for *data/creature.py* to import `sqlite3` itself — unless someday it's necessary to call another `sqlite3` method that isn't a method of the `conn` or `curs` objects.

Again, these changes should goose Uvicorn into reloading everything. From now on, testing with any of the methods that you've seen so far (Httpie and friends, or the automated `/docs` forms) will show data that persist. If you add a creature, it will be there the next time you get all of them.

Let's do the same for explorers in Example 10-5.

Example 10-5. Add database configuration to data/explorer.py

```
from .init import conn, curs
from ..model.explorer import Explorer

curs.execute("""create table if not exists explorer(
                name text primary key,
                nationality text)""")

def row_to_model(row: tuple) -> Explorer:
    return Explorer(name=row[0], nationality=row[1])
```

```
def model_to_dict(explorer: Explorer) -> dict:
    return explorer.dict() if explorer else None

def get_one(name: str) -> Explorer:
    qry = "select * from explorer where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    return row_to_model(curs.fetchone())

def get_all() -> list[Explorer]:
    qry = "select * from explorer"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(explorer: Explorer) -> Explorer:
    qry = """insert into explorer (name, nationality)
             values (:name, :nationality)"""
    params = model_to_dict(explorer)
    res = curs.execute(qry, params)
    return get_one(explorer.name)

def modify(name: str, explorer: Explorer) -> Explorer:
    qry = """update explorer
             set nationality=:nationality, name=:name
             where name=:name0"""
    params = model_to_dict(explorer)
    params["name0"] = explorer.name
    res = curs.execute(qry, params)
    explorer2 = get_one(explorer.name)
    return explorer2

def delete(explorer: Explorer) -> bool:
    qry = "delete from explorer where name = :name"
    params = {"name": explorer.name}
    res = curs.execute(qry, params)
    return bool(res)
```

Test!

That’s a lot of code with no tests. Does everything work? I’d actually be surprised if it all did. So let’s set up some tests:

Make these subdirectories under the *test* directory: * *unit*: Within a layer * *full*: Across all layers

Which type should you write and run first? Most people write automated unit tests first; they’re smaller, and all the other layer pieces may not exist yet. In this book, development has been top-down, and we’re now completing the last layer. Also, we did manual tests (with Httpie and friends) in the Web and Service chapters. Those helped to expose bugs and omissions quickly; automated tests ensure that you don’t keep making the same errors later. So, I’d recommend:

- Some manual tests as you’re first writing the code.
- Unit tests once you’ve fixed Python syntax errors.
- Full tests once you have a full data flow across all layers.

Full tests

These call the web endpoints, which take the code elevator down through Service to Data, and back up again. Sometimes these are called *end-to-end* or *contract* tests.

Get all explorers

Dipping a toe in the test waters, not yet knowing if they’re infested with piranhas, is brave volunteer Example 10-6.

Example 10-6. Test get all explorers

```
$ http localhost:8000/explorer
HTTP/1.1 405 Method Not Allowed
allow: POST
content-length: 31
content-type: application/json
date: Mon, 27 Feb 2023 20:05:18 GMT
server: uvicorn

{
    "detail": "Method Not Allowed"
}
```

Eek! What happened?

Oh. I asked for `/explorer`, not `/explorer/`, and there’s no `GET`-method path function for the URL `/explorer` (with no final slash). In *web/explorer.py*, the path decorator for the `get_all()` path function is:

```
@router.get("/")
```

That, plus the earlier code:

```
router = APIRouter(prefix = "/explorer")
```

means this `get_all()` path function serves a URL containing `/explorer/`.

Example 20-7 happily shows that you can have more than one path decorator per path function.

Example 10-7. Add a non-slash path decorator for the get_all() path function

```
@router.get("")
@router.get("/")
def get_all() -> list[Explorer]:
    return service.get_all()
```

Test with both URLs in Examples 10-8 and 10-9:

Example 10-8. Test the non-slash endpoint.

```
$ http localhost:8000/explorer
HTTP/1.1 200 OK
content-length: 2
content-type: application/json
date: Mon, 27 Feb 2023 20:12:44 GMT
server: uvicorn

[]
```

Example 10-9. Test the slash endpoint

```
$ http localhost:8000/explorer/
HTTP/1.1 200 OK
content-length: 2
content-type: application/json
date: Mon, 27 Feb 2023 20:14:39 GMT
server: uvicorn

[]
```

Now that both of these work, create an explorer, and retry the get all test after. Example 10-10 will attempt this, but with a plot twist:

Example 10-10. Test explorer creation, with an input error

```
$ http post localhost:8000/explorer name="Beau Buffette",
natinality="United States"
HTTP/1.1 422 Unprocessable Entity
content-length: 95
content-type: application/json
```

```
date: Mon, 27 Feb 2023 20:17:45 GMT
server: uvicorn

{
    "detail": [
        {
            "loc": [
                "body",
                "nationality"
            ],
            "msg": "field required",
            "type": "value_error.missing"
        }
    ]
}
```

I misspelled `nationality`, although my speling is usually impeckable. Pydantic caught this in the Web layer, returning a 422 HTTP status code and a description of the problem. Generally, if FastAPI returns a 422, the odds are that Pydantic fingered the perpetrator. The `"loc"` part says where the error occurred: the field `"nationality"` was missing, because I'm such an inept typist.

Fix the spelling and retest in Example 10-11:

Example 10-11. Create with corrected value

```
$ http post localhost:8000/explorer name="Beau Buffette"
nationality="United States"
HTTP/1.1 201 Created
content-length: 55
content-type: application/json
date: Mon, 27 Feb 2023 20:20:49 GMT
server: uvicorn

{
    "name": "Beau Buffette,",
    "nationality": "United States"
}
```

This time it returned a `201` status code, which is traditional when a resource is created (all *2xx* status codes are considered to indicate success, with plain `200` being the most generic). The response also contained the JSON version of the `Explorer` object that was just created.

Now back to the initial test: will Beau turn up in the get all explorers test?

Example 10-12 answers this burning question:

Example 10-12. Did the latest create() work?

```
$ http localhost:8000/explorer
HTTP/1.1 200 OK
content-length: 57
content-type: application/json
date: Mon, 27 Feb 2023 20:26:26 GMT
server: uvicorn

[
    {
        "name": "Beau Buffette",
        "nationality": "United States"
    }
]
```

Yay.

Get One Explorer

Now, what happens if you try to look up Beau with the get *one* endpoint (Example 10-13)?

Example 10-13. Test the get one endpoint

```
$ http localhost:8000/explorer/"Beau Buffette"
HTTP/1.1 200 OK
content-length: 55
content-type: application/json
date: Mon, 27 Feb 2023 20:28:48 GMT
server: uvicorn

{
    "name": "Beau Buffette,",
    "nationality": "United States"
}
```

I used the quotes to preserve that space between the first and last names. In URLs, you could also use `Beau%20Buffette`; the `%20` is the hex code for the space character in ASCII.

Missing and Duplicate Data

There are two main error classes that I've ignored so far:

- *Missing data*: If you try to get, modify, or delete an explorer by a name that isn’t in the database.

- *Duplicate data*: If you try to create an explorer with the same name more than once.

So, what if I ask for a non-existent or duplicate explorer? So far, the code has been too optimistic, and exceptions will bubble up from the abyss.

Our friend Beau was just added to the database. Imagine his evil clone (who shares his name) plots to replace him some dark night, using Example 10-14:

Example 10-14. Duplicate error: try to create an explorer more than once

```
$ http post localhost:8000/explorer name="Beau Buffette"
nationality="United States"
HTTP/1.1 500 Internal Server Error
content-length: 3127
content-type: text/plain; charset=utf-8
date: Mon, 27 Feb 2023 21:04:09 GMT
server: uvicorn

Traceback (most recent call last):
  File ".../starlette/middleware/errors.py", line 162, in call
... (lots of confusing innards here) ...
  File ".../service/explorer.py", line 11, in create
    return data.create(explorer)
           ^^^^^^^
  File ".../data/explorer.py", line 37, in create
    curs.execute(qry, params)
sqlite3.IntegrityError: UNIQUE constraint failed: explorer.name
```

I omitted most of the lines in that error trace (and replaced some parts with ellipses), because it contained mostly internal calls made by FastAPI and the underlying Starlette.

But that last line: a SQLite exception in the Web layer! Where is the fainting couch?

Right on the heels of this, yet another horror in Example 10-15: a missing explorer.

Example 10-15. Get a non-existent explorer

```
$ http localhost:8000/explorer/"Beau Buffalo"
HTTP/1.1 500 Internal Server Error
content-length: 3282
content-type: text/plain; charset=utf-8
```

```
date: Mon, 27 Feb 2023 21:09:37 GMT
server: uvicorn

Traceback (most recent call last):
  File ".../starlette/middleware/errors.py", line 162, in call
... (many lines of ancient cuneiform) ...
  File ".../data/explorer.py", line 11, in row_to_model
    name, nationality = row
    ^^^^^^^
TypeError: cannot unpack non-iterable NoneType object
```

What's a good way to catch these at the bottom (Data) layer, and communicate the details to the top (Web)? Possibilities include:

- Let SQLite cough up a hairball (exception) and deal with it in the Web layer.
  - But: this mixes the layers, which is Bad. The Web layer should not know anything about specific databases.
- Make every function in the Service and Data layers return `Explorer | None` where they used to return `Explorer`. Then a `None` indicates failure. (You can shorten this by defining `OptExplorer = Explorer | None` in *model/explorer.py*.)
  - But: it may have failed for more than one reason, and you might want details. And this requires lots of code editing.
- Define exceptions for `Missing` and `Duplicate` data, including details of the problem. These will flow up through the layers with no code changes until the Web path functions catch them. They're also application-specific rather than database-specific, preserving the sanctity of the layers.
  - But: actually, I like this one, si it goes in Example 10-16.

Example 10-16. Define a new top-level errors.py

```python
class Missing(Exception):
    def __init__(self, msg:str):
        self.msg = msg

class Duplicate(Exception):
    def __init__(self, msg:str):
        self.msg = msg
```

Each of these has a `msg` string attribute that can inform the higher-level code what happened.

To implement this, in Example 10-7, have *data/init.py* import the DB-API exception that SQLite would raise for a duplicate:

Example 10-17. Add a SQLite exception import into data/init.py

```python
from sqlite3 import connect, IntegrityError
```

Import and catch this error in Example 10-18:

Example 10-18. Modify data/explorer.py to catch and raise these exceptions

```python
from .init import (conn, curs, IntegrityError)
from ..model.explorer import Explorer
from ..errors import Missing, Duplicate

curs.execute("""create table if not exists explorer(
                name text primary key,
                nationality text)""")

def row_to_model(row: tuple) -> Explorer:
    name, nationality = row
    return Explorer(name=name, nationality=nationality)

def model_to_dict(explorer: Explorer) -> dict:
    return explorer.dict()

def get_one(name: str) -> Explorer:
    qry = "select * from explorer where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    row = curs.fetchone()
    if row:
        return row_to_model(row)
    else:
        raise Missing(msg=f"Explorer {name} not found")

def get_all() -> list[Explorer]:
    qry = "select * from explorer"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(explorer: Explorer) -> Explorer:
    if not explorer: return None
    qry = """insert into explorer (name, nationality) values
             (:name, :nationality)"""
    params = model_to_dict(explorer)
    try:
        curs.execute(qry, params)
    except IntegrityError:
        raise Duplicate(msg=
            f"Explorer {explorer.name} already exists")
    return get_one(explorer.name)

def modify(name: str, explorer: Explorer) -> Explorer:
    if not (name and explorer): return None
    qry = """update explorer
             set nationality=:nationality, name=:name
             where name=:name0"""
    params = model_to_dict(explorer)
    params["name0"] = explorer.name
    curs.execute(qry, params)
    if curs.rowcount == 1:
        return get_one(explorer.name)
    else:
        raise Missing(msg=f"Explorer {name} not found")

def delete(name: str):
    if not name: return False
    qry = "delete from explorer where name = :name"
    params = {"name": name}
    curs.execute(qry, params)
    if curs.rowcount != 1:
        raise Missing(msg=f"Explorer {name} not found")
```

This drops the need to declare that any functions return `Explorer | None` or `Optional[Explorer]`. You only indicate type hints for normal return types, not exceptions. Because exceptions flow upward independent of the call stack until someone catches them, for once I don't have to change anything in the Service layer. But here's the new *web/explorer.py* in Example 10-19, with exception handlers and appropriate HTTP status code returns:

Example 10-19. Handle `Missing` and `Duplicate` exceptions in web/explorer.py

```
from fastapi import APIRouter, HTTPException
from ..model.explorer import Explorer
from ..service import explorer as service
from ..errors import Duplicate, Missing

router = APIRouter(prefix = "/explorer")

@router.get("")
@router.get("/")
def get_all() -> list[Explorer]:
    return service.get_all()
```

```
@router.get("/{name}")
def get_one(name) -> Explorer:
    try:
        return service.get_one(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.post("", status_code=201)
@router.post("/", status_code=201)
def create(explorer: Explorer) -> Explorer:
    try:
        return service.create(explorer)
    except Duplicate as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.patch("/")
def modify(name: str, explorer: Explorer) -> Explorer:
    try:
        return service.modify(name, explorer)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.delete("/{name}", status_code=204)
def delete(name: str):
    try:
        return service.delete(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)
```

Test these changes in Example 10-20:

Example 10-20. Test get one of non-existing explorer again, with new `Missing` exception

```
$ http localhost:8000/explorer/"Beau Buffalo"
HTTP/1.1 404 Not Found
content-length: 44
content-type: application/json
date: Mon, 27 Feb 2023 21:11:27 GMT
server: uvicorn

{
    "detail": "Explorer Beau Buffalo not found"
}
```

Good. Now, try the evil clone attempt again in Example 10-21:

Example 10-21. Test duplicate fix

```
$ http post localhost:8000/explorer name="Beau Buffette"
nationality="United States"
HTTP/1.1 404 Not Found
content-length: 50
content-type: application/json
date: Mon, 27 Feb 2023 21:14:00 GMT
server: uvicorn

{
    "detail": "Explorer Beau Buffette already exists"
}
```

The missing checks would also apply to the modify and delete endpoints. You can try writing similar tests for them.

Unit Tests

(TO DO: move this section up, after missing/duplicate stuff)

These only deal with the Data layer, checking the database calls and SQL syntax. I’ve put this section after the full tests because I wanted to have the `Missing` and `Duplicate` exceptions already defined, explained , and coded into *data/creature.py*. Example 10-6 lists the test script *test/unit/data/test_creature.py*. Some things to note:

- I set the environment variable `CRYPTID_SQLITE_DATABASE` to `":memory:"` *before* importing `init` or `creature` from `data`. This value makes SQLite work completely in memory, not stomping any existing database file, or even creating a file on disk. It’s checked in *data/init.py* when that module is first imported.
- The *fixture* named `sample` is passed to the functions that need a `Creature` object.
- The tests run in order. In this case, the same database stays up the whole time, instead of being reset between functions. The reason was to allow changes from previous functions to persist. With pytest, the a fixture can have:
    - *Function* scope (the default): It’s called anew before every test function.

    - *Session* scope: It’s called only once, at the start.

- Some tests force the `Missing` or `Duplicate` exceptions, and verify that they caught them.

So, each of the tests below gets a brand new, unchanged `Creature` object named `sample`.

Example 10-22. Unit tests for data/creature.py

```
import os
import pytest
from ....model.creature import Creature
from ....error import Missing, Duplicate

# Set this before data.init import below
os.environ["CRYPTID_SQLITE_DB"] = ":memory:"
from ...data import init, creature

@pytest.fixture
def sample() -> Creature:
    return Creature(name="yeti",
        description="Abominable Snowman",
        location="Himalayas")

def test_create(sample):
    resp = creature.create(sample)
    assert resp == sample

def test_create_duplicate(sample):
    with pytest.raises(Duplicate):
        resp = creature.create(sample)

def test_get_exists(sample):
    resp = creature.get_one(sample.name)
    assert resp == sample

def test_get_missing():
    with pytest.raises(Missing):
        resp = creature.get_one("boxturtle")

def test_modify(sample):
    creature.location = "Sesame Street"
    resp = creature.modify(sample.name, sample)
    assert resp == sample

def test_modify_missing():
    bob: Creature = Creature(name="bob",
        description="some guy", location="somewhere")
    with pytest.raises(Missing):
        resp = creature.modify(bob.name, bob)

def test_delete(sample):
    resp = creature.delete(sample.name)
    assert resp is None

def test_delete_missing(sample):
    with pytest.raises(Missing):
        resp = creature.delete(sample.name)
```

Hint: you can make your own version of *test/unit/data/test_explorer.py*.

Review

This chapter presented a simple data handling layer, with a few trips up and down the layer stack as needed. Chapter 12 contains unit tests for each layer, as well as cross-layer integration and full end-to-end tests. Chapter 17 goes into more database depth and detailed examples.

Chapter 11. Authentication and Authorization

Respect mah authoritay!

—Eric Cartman, _South Park_

Preview

Sometimes a website is wide open (*read-only*) and any visitor can visit any page. But if any of the site's content may be modified, some endpoints will be restricted to certain people or groups. If anyone could alter pages on Amazon, imagine the odd items that would show up, and the amazing sales some people would suddenly get. Unfortunately, it's human nature — for some humans — to take advantage of the rest, who pay a hidden tax for their activities.

Should we leave our cryptid site open for any users to access any endpoint? No! Almost any sizable web service eventually needs to deal with:

- *Authentication* (*authn*): Who are you?
- *Authorization* (*authz*): What do you want?

Should the auth code have its own new layer, say between Web and Service? Or should everything be handled by the Web or Service layer itself? This chapter dips into auth techniques and where to put them.

Often, descriptions of web security seem more confusing than they need to be. Attackers can be really, really sneaky, and countermeasures may not be simple.

> **NOTE**
>
> As I’ve mentioned more than once, the official FastAPI documentation is excellent. Try the security section if this book chapter doesn’t quite explain things as well as you’d like.

So, let’s take it in steps. I’ll start with simple techniques that are only intended to hook auth into a web endpoint for testing, but would not stand up in a public website.

Interlude 1: Do You Need Authentication?

Again, *authentication* is concerned with *identity*: who are you?. To implement this, somewhere there needs to a mapping of some secret information to a unique identity. There are many ways to do this, with *many* variations of complexity. Let’s start small and work up.

Often, books and articles on web development jump right away into the details of authentication and authorization, sometimes muddling them. They sometimes skip the first question: do you really need either?

You could allow completely anonymous access to all of your website’s pages. But that would leave you open to exploits like denial of service attacks. Although some protections like rate limits can be implemented outside the web server (see Chapter 13), almost all public API providers require at least some authentication.

Beyond security, websites want to know how they’re doing:

- How many unique visitors?
- What are the most popular pages?

- Do some changes increase views?
- What page sequences are common?

These require authentication of specific visitors. Otherwise you can only get total counts.

> **NOTE**
>
> If your site needs authentication or authorization, then access to it should be encrypted (HTTPS vs. HTTP), to prevent attackers from extracting secret data from plain-text. See Chapter 13 for details on setting up HTTPS.

Authentication Methods

There are way too many web authentication methods and tools:

- *Username/email and password*: Using classic HTTP Basic and Digest Authentication.
- *API Key*: An opaque long string, with an accompanying *secret*.
- *OAuth2*: A set of standards for authentication and authorization.
- *JWT (JavaScript Web Tokens)*: An encoding format containing cryptographically signed user information

In the following two sections, I'll review the first two methods and show how you would traditionally implement them. But I'll stop before filling out the API and database code. Instead, the sections following those fully implement a more modern scheme with OAuth2 and JWT.

Global Authentication: Shared Secret

The very simplest authentication method is to pass a secret that's normally only known by the web server. If it matches, you're in. This isn't safe if your API site is exposed to the public with HTTP instead of HTTPS. If it's hidden behind a front-end site that is itself open, they could communicate using a shared constant secret. But if your front-end site is hacked, then darn. But let’s see how FastAPI handles simple authentication.

Make a new top-level file called *auth.py*. Make sure that you don’t have another FastAPI server still running from one of those ever-changing *main.py* files from previous chapters. Example 11-1 implements a server that just returns whatever `username` and `password` were sent to it using HTTP Basic Authentication — a method from the original days of the Web.

Example 11-1. Use HTTP Basic Auth to get user info: auth.py

```
import uvicorn
from fastapi import Depends, FastAPI
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()

basic = HTTPBasic()

@app.get("/who")
def get_user(
    creds: HTTPBasicCredentials = Depends(basic)):
    return {"username": creds.username, "password": creds.password}

if __name__ == "__main__":
    uvicorn.run("auth:app", reload=True)
```

In Example 11-2, tell Httpie to make this Basic Auth request (this requires the arguments `-a` *name:password*). Here, let’s use the name “me” and the password “secret”:

Example 11-2. Test with Httpie

```
$ http -q -a me:secret localhost:8000/who
{
    "password": "secret",
    "username": "me"
}
```

Testing with the Requests package in Example 11-3 is similar, using the `auth` parameter:

Example 11-3. Test with Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/who",
    auth=("me", "secret"))
>>> r.json()
```

```
{'username': 'me', 'password': 'secret'}
```

You can also test it with the automatic docs page (*http://localhost:8000/docs*), shown in Figure 11-1:

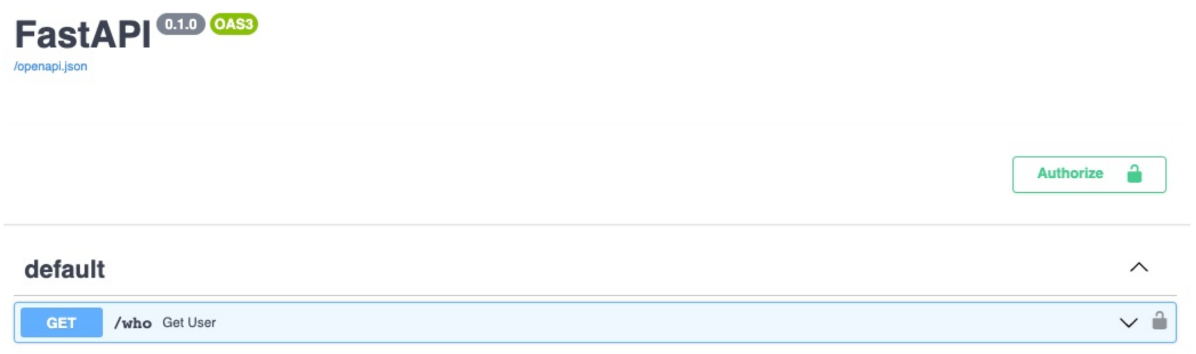


Figure 11-1. Docs page for simple authentication

Click on that down arrow on the right, then the Try It Out button, and then the Execute button. You'll see a form requesting the username and password. Type anything. It will hit that server endpoint and show those values in the response.

These tests showed that you can get a username and password to the server and back (although none of these actually checked anything). Something in the server needs to verify that this name and password match some approved values. So, in Example 11-4, I'll include a single secret username and password in the web server. The username and password that you pass in now needs to match them (each is a *shared secret*), or you'll get an exception. The HTTP status code `401` is officially called `Unauthorized`, but it really means *unauthenticated*.

NOTE

Instead of memorizing all the HTTP status codes, you can import FastAPI's `status` module (which itself is imported directly from Starlette). So you can use the more explanatory `status_code=HTTP_401_UNAUTHORIZED` in the code below instead of a plain `status_code=401`.

Example 11-4. Add a secret username and password to auth.py

```
import uvicorn
from fastapi import Depends, FastAPI
from fastapi.security import HTTPBasic, HTTPBasicCredentials
```

```python
app = FastAPI()

secret_user: str = "newphone"
secret_password: str = "whodis?"

basic: HTTPBasicCredentials = HTTPBasic()

@app.get("/who")
def get_user(
    creds: HTTPBasicCredentials = Depends(basic)) -> dict:
    if (creds.username == secret_user and
        creds.password == secret_password):
        return {"username": creds.username,
            "password": creds.password}
    raise HTTPException(status_code=401, detail="Hey!")

if __name__ == "__main__":
    uvicorn.run("auth:app", reload=True)
```

Misguessing the username and password will earn a mild `401` rebuke in Example 11-5:

Example 11-5. Test with Httpie and mismatched username/passsord

```
$ http -a me:secret localhost:8000/who
HTTP/1.1 401 Unauthorized
content-length: 17
content-type: application/json
date: Fri, 03 Mar 2023 03:25:09 GMT
server: uvicorn

{
    "detail": "Hey!"
}
```

Using the magic combination returns them, as before, in Example 11-6:

Example 11-6. Test with Httpie and correct username/passsord

```
$ http -q -a newphone:whodis? localhost:8000/who
{
    "password": "whodis?",
    "username": "newphone"
}
```

Simple Individual Authentication

The previous section showed how you could use a shared secret to control access. It’s a broad approach, not very secure. And it doesn’t tell you anything about the individual visitor, just that he or she (or a sentient AI) knows the secret.

Many websites want to:

- Define individual visitors in some way.
- Identify specific visitors as they access certain endpoints (authentication).
- Possibly assign different permissions to some visitors and endpoints (authorization).
- Possibly save specific information per visitor (interests, purchases, and so on).

If your “visitors” are humans, you may want them to provide a username or email and a password. If they’re external programs, you may want them to provide an API key and secret.

NOTE

From here on, I’ll user just *username* to refer either to a user-selected name or an email.

To authenticate real individual users instead of one fake one, you’ll need to do a bit more:

- Pass the user values (name and password) to the API server endpoints as HTTP headers.
- Use HTTPS instead of HTTP, to avoid anyone snooping the text of these headers.
- *Hash* the password to a different string. The result is not “de-hashable" — you can’t derive the original password from its hash.
- Make a real database store a `User` database table containing the username and the hashed password (never the original plain text password).
- Hash the newly input password and compare the result with the hashed

password in the database.

- If the username and hashed password match, pass the matching `User` object up the stack. If no match, return `None` or raise an exception.
- In the Service layer, fire off any metrics/logging/whatever that are relevant to individual user authentication.
- In the Web layer, send the authenticated user info to any functions that require it.

I’ll show you how to do all these things in the following sections, using recent tools like OAuth2 and JWT.

Fancier Individual Authentication

If you want to authenticate individuals, then you have to store some individual information somewhere — like a database containing records with at least a key (username or API key), and a secret (password or API secret). Your website visitors will provide these when accessing protected URLs, and you need something in the database to match them with.

The official FastAPI security docs (introductory and advanced) have top-down descriptions of how to set up authentication for multiple users, using a local database. But, in their example web functions, they fake the actual database access.

Here, I’ll do the opposite: starting at the Data layer and working up. We’ll define how a user/visitor is defined, stored, and accessed. Then we’ll work up to the Web layer, and how user identification is passed in, evaluated, and authenticated.

OAuth2

> *OAuth 2.0, which stands for “Open Authorization”, is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.*
>
> —auth0.com

In the early trusting web days, you could provide your login name and password of a website (let’s call it B) to another website (A, of course) and let it access stuff on B for you. This would give A *full access* to B, although it was trusted to access only what it was supposed to. Examples of B and resources were things like Twitter followers, Facebook friends, email contacts, and so on. Of course, this couldn’t last long, so various companies and groups got together to define OAuth. It was originally designed only to allow website A to access specific (not all) resources on website B.

OAuth2 is a popular but complex *authorization* standard, with uses beyond the A/B example above. There are many explanations of it, from light to heavy.

NOTE

There used to be an OAuth1, but it isn’t used anymore. Some of the original OAuth2 recommendations are now deprecated (computerese for *don’t use them*). On the horizon are OAuth2.1 and, even further into the mist, TXAuth.

OAuth offers various flows for different circumstances. I’ll use the *authorization code flow* here. This section will walk through an implementation, one average-sized step at a time.

First, you’ll need to install some third party Python packages:

- JWT handling: `pip install python-jose[cryptography]`
- Secure password handling: `pip install passlib`
- Form handling: `pip install python-multipart`

The following sections start with the user data model and database management, and work up the familiar layers to the Service and Web, where OAuth pops up.

User Model

Let’s start with a very minimal user model definitions in Example 11-7. These will be used in all layers.

Example 11-7. User definition: model/user.py

```
from pydantic import BaseModel
```

```
class User(BaseModel):
    name: str
    hash: str
```

A `User` object contains an arbitrary `name` plus a `hash` string (the hashed password, not the original plain text password), and is what's saved in the database. We'll need both to authenticate a visitor.

User Data Layer

Example 11-8 contains the user database code.

> **NOTE**
>
> The code creates `user` (active users) and `xuser` (deleted users) tables. Often developers add a Boolean `deleted` field to a user table to indicate the user is no longer active, without actually deleting the record from the table. I prefer moving the deleted user's data to another table. This avoids repetetive checking of the `deleted` field in all user queries. It can also help speed up queries: making an index for a *low cardinality* field like a Boolean does no good.

Example 11-8. Data layer: data/user.py

```
from model.user import User
from .init import (conn, curs, get_db, IntegrityError)
from error import Missing, Duplicate

curs.execute("""create table if not exists
                user(
                  name text primary key,
                  hash text)""")
curs.execute("""create table if not exists
                xuser(
                  name text primary key,
                  hash text)""")

def row_to_model(row: tuple) -> User:
    name, hash = row
    return User(name=name, hash=hash)

def model_to_dict(user: User) -> dict:
    return user.dict()

def get_one(name: str) -> User:
    qry = "select * from user where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    row = curs.fetchone()
    if row:
        return row_to_model(row)
    else:
        raise Missing(msg=f"User {name} not found")

def get_all() -> list[User]:
    qry = "select * from user"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(user: User, table = "user" | "xuser" = "user"):
    """Add <user> to user or xuser table"""
    qry = f"""insert into {table}
        (name, hash)
        values
        (:name, :hash)"""
    params = model_to_dict(user)
    try:
        curs.execute(qry, params)
    except IntegrityError:
        raise Duplicate(msg=
            f"{table}: user {user.name} already exists")

def modify(name: str, user: User)  -> User:
    qry = """update user set
             name=:name, hash=:hash
             where name=:name0"""
    params = {
        "name": user.name,
        "hash": user.hash,
        "name0": name}
    curs.execute(qry, params)
    if curs.rowcount == 1:
        return get_one(user.name)
    else:
        raise Missing(msg=f"User {name} not found")

def delete(name: str) -> None:
    """Drop user with <name> from user table, add to xuser table"""
    user = get_one(name)
    qry = "delete from user where name = :name"
    params = {"name": name}
    curs.execute(qry, params)
    if curs.rowcount != 1:
        raise Missing(msg=f"User {name} not found")
    create(user, table="xuser")
```

User Fake Data Layer

The module in Example 11-9 is used in tests that exclude the database but need some user data.

Example 11-9. Fake layer: fake/user.py

```
from model.user import User
from error import Missing, Duplicate

# (no hashed password checking in this module)
fakes = [
    User(name="kwijobo",
         hash="abc"),
    User(name="ermagerd",
         hash="xyz"),
    ]

def find(name: str) -> User | None:
    for e in fakes:
        if e.name == name:
            return e
    return None

def check_missing(name: str):
    if not find(name):
        raise Missing(msg=f"Missing user {name}")

def check_duplicate(name: str):
    if find(name):
        raise Duplicate(msg=f"Duplicate user {name}")

def get_all() -> list[User]:
    """Return all users"""
    return fakes

def get_one(name: str) -> User:
    """Return one user"""
    check_missing(name)
    return find(name)

def create(user: User) -> User:
    """Add a user"""
    check_duplicate(user.name)
    return user

def modify(name: str, user: User) -> User:
    """Partially modify a user"""
    check_missing(name)
    return user

def delete(name: str) -> None:
    """Delete a user"""
    check_missing(name)
    return None
```

User Service Layer

Example 11-10 defines the service layer for users. A difference from the other Service layer modules is the addition of OAuth2 and JWT functions. I think it's cleaner to have them here than in the Web layer, though a few OAuth2 Web-layer functions are in the upcoming *web/user.py*. The CRUD functions are still passthroughs for now, but could be flavored to taste with metrics in the future. Notice that, like the creature and explorer services, this supports runtime use of either the fake or real data layers to access User data.

Example 11-10. Service layer: service/user.py

```
from datetime imoport timedelta
import os
from jose import jwt
from model.user import User

if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import user as data
else:
    from data import user as data

# --- New auth stuff

from passlib.context import CryptContext

# Change SECRET_KEY for production!
SECRET_KEY = "keep-it-secret-keep-it-safe"
ALGORITHM = "HS256"
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def verify_password(plain: str, hash: str) -> bool:
    """Hash <plain> and compare with <hash> from the database"""
    return pwd_context.verify(plain, hash)

def get_hash(plain: str) -> str:
    """Return the hash of a <plain> string"""
    return pwd_context.hash(plain)

def get_jwt_username(token:str) -> str | None:
```

```python
    """Return username from JWT access <token>"""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        if not (username := payload.get("sub")):
            return None
    except jwt.JWTError:
        return None
    return username

def get_current_user(token: str) -> User | None:
    """Decode an OAuth access <token> and return the User"""
    if not (username := get_jwt_username(token)):
        return None
    if (user := lookup_user(username)):
        return user
    return None

def lookup_user(name: str) -> User | None:
    """Return a matching User fron the database for <name>"""
    if (user := data.get(username)):
        return user
    return None

def auth_user(name: str, plain: str) -> User | None:
    """Authenticate user <name> and <plain> password"""
    if not (user := lookup_user(name)):
        return None
    if not verify_password(plain, user.hash):
        return None
    return user

def create_access_token(data: dict,
    expires: timedelta | None = None
):
    """Return a JWT access token"""
    src = data.copy()
    now = datetime.utcnow()
    expires = timedelta(minutes=15) if not expires
    src.update({"exp": now + expires})
    encoded_jwt = jwt.encode(src, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

# --- CRUD passthrough stuff

def get_all() -> list[User]:
    return data.get_all()

def get_one(name) -> User:
    return data.get_one(name)
```

```
def create(user: User) -> User:
    return data.create(user)

def modify(name: str, user: User) -> User:
    return data.modify(name, user)

def delete(name: str) -> None:
    return data.delete(name)
```

User Web Layer

Example 11-11 defines the base user module in the Web layer. It uses the new auth code from the *service/user.py* module in Example 11-10.

Example 11-11. Web layer: web/user.py

```
import os
from fastapi import APIRouter, HTTPException
from model.user import User
if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import user as service
else:
    from service import user as service
from error import Missing, Duplicate

router = APIRouter(prefix = "/user")

# --- new auth stuff

# This dependency makes a post to "/user/token"
# (from a form containing a username and password)
# return an access token.
oauth2_dep = OAuth2PasswordBearer(tokenUrl="token")

def unauthed():
    raise HTTPException(
        status_code=401,
        detail="Incorrect username or password",
        headers={"WWW-Authenticate": "Bearer"},

# This endpoint is directed to by any call that has the
@ oauth2_dep() dependency:
@router.post("/token")
async def create_access_token(
    form_data: OAuth2PasswordRequestForm =  Depends()
):
    """Get username and password from OAuth form,
        return access token"""
    user = service.auth_user(form_data.username, form_data.password)
```

```python
    if not user:
        unauthed()
    expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = service.create_access_token(
        data={"sub": user.username}, expires=expires
    )
    return {"access_token": access_token, "token_type": "bearer"}

@app.get("/token")
def get_access_token(token: str = Depends(oauth2_token)) -> dict:
    """Return the current access token"""
    return {"token": token}

# --- previous CRUD stuff

@router.get("/")
def get_all() -> list[User]:
    return service.get_all()

@router.get("/{name}")
def get_one(name) -> User:
    try:
        return service.get_one(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.post("/", status_code=201)
def create(user: User) -> User:
    try:
        return service.create(creature)
    except Duplicate as exc:
        raise HTTPException(status_code=409, detail=exc.msg)

@router.patch("/")
def modify(name: str, user: User) -> User:
    try:
        return service.modify(name, user)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.delete("/{name}")
def delete(name: str) -> None:
    try:
        return service.delete(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)
```

Test!

The unit and full tests for this new user component are very similar to those that you'll already seen for creatures and explorers. Rather than using the ink and paper here[1], you can view them at this book's accompanying website.

Top Layer

The previous section defined a new `router` variable for URLs starting with `/user`, so Example 11-X adds this subrouter.

Example 11-12. Top layer: main.py

```
from fastapi import FastAPI
from web import explorer, creature, user

app = FastAPI()
app.include_router(explorer.router)
app.include_router(creature.router)
app.include_router(user.router)
```

When uvicorn autoreloads, the `/user/...` endpoints should now be available.

That was fun, for some stretched definition of fun. Given all the user code that was just created, let's give it something to do.

Authentication Steps

To review that heap of code from the previous sections:

- If an endpoint has the dependency `oauth2_dep()` (in *web/user.py*), a form containing username and password fields is generated and sent to the client.
- After the client fills out and submits this form, the username and password (hashed with the same algorithm as those already stored in the local database) are matched against the local database.
- If a match, an access token is generated (in JWT format) and returned.
- This access token is passed back to the web server as an `Authorization` HTTP header in subsequent requests. This JWT token is decoded on the local server to the username and other details. This name does not need to be looked up in the database again.

- The username is authenticated, and the server can do whatever it likes with it.

What can the server do with this hard-won authentication information?

- Generate metrics (this user, this endpoint, this time) to help study what's being viewed, by whom, for how long, and so on.
- Save user-specific information.

JWT (JSON Web Token)

This section contains some details on the JWT. You really don't need them to use all the earlier code in this chapter, but if you're a little curious…

A JWT is an encoding scheme, not an authentication method. The low-level details are defined in RFC 7519. It can be used to convey authentication informatiom for OAuth2 (and other methods), and I'll show that here.

A JWT is a readable string with three dot-separated sections:

- *Header*: Encryption algorithm used, and token type
- *Payload*: …
- *Signature*: …

Each section consists of a JSON string, encoded in Base 64 URL format. Here's an example (split at the dots to fit on this page):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MD
IyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

As a plain ASCII string that's also safe to use in URLs, it can be passed to web servers as part of the URL, a query parameter, HTTP header, cookie, and so on.

JWT advantages:

- Avoids database lookup

Because no solution fits every problem, disadvantages:

- Lack of database lookup means that you can’t detect a revoked authorization directly.

Third-Party Authentication: OIDC (OpenId Connect)

You’ll often see websites that let you log in with some id and password, or let you log in via your account at a different site, like Google, Facebook/Meta, LinkedIn, or many others. These frequently use a standard called OpenId Connect, which is built atop OAuth2. When you connect to an external OIDC-enabled site, you’ll get back an OAuth2 access token (as in the examples in this chapter), but also an *id token*.

The official FastAPI docs don’t include example code for integration with OIDC, but if you want to try it, there are some third-party packages (FastAPI-specific and more generic) that will save time over rolling your own implementation:

- fastapi-oidc
- fastapi-third-party-auth
- fastapi_resource_server
- oauthlib
- oic
- oidc-client
- oidc-op
- openid-connect

This link includes multiple code examples, and a comment from tiangelo (Sebastián Ramírez) that FastAPI-OIDC examples will be included in the official docs and tutorials in the future.

Authorization

Authentication handles the *who* (identity), and authorization handles the *what*:

which resources (web endpoints) are you allowed to access, and in what way? The number of combinations of *who* and *what* can be very large.

In this book, explorers and creatures have been the main resources. Looking up an explorer, or listing all of them, would normally be more “open” than adding or modifying an existing one. If the website is supposed to be a reliable interface to some data, then write access should be more limited than read access. Because, grr, people.

If every endpoint is completely open, you don’t need authorization, and can skip this section. The simplest authorization could be a simple binary *admin* versus not — for the examples in this book, you might require admin authorization to add, delete, or modify an explorer or creature. If your database had lots of entries, you might also want to limit the `get_all()` functions with further permissions for non-admins. As the website gets more complex, the permissions might become more fine-grained.

Let’s look at a progression of authorization cases. I’ll use the `User` table (where the `name` can be an email, username, or API key) (“pair” tables are the relational database way of matching entries from two separate tables):

- If you only want to track admin visitors, and leave the rest anonymous:
  - An `Admin` table of authenticated user names. In the code examples earlier in this chapter, you’d look up the name from the `Admin` table, and if matched, compare the hashed passwords from the `User` table.
- If *all* visitors should be authenticated, but you only need to authorize admins for some endpoints:
  - Authenticate everyone as in the earlier examples (from the `User` table), then check the `Admin` table to see if this user is also an admin.
- For more than one type of permission (such as read-only, read, write):
  - A `Permission` definition table.
  - A `UserPermission` table that pairs users and permissions. This is sometimes called an *access control list*.

- If permission combinations are complex, add a level and define *roles* (independent sets of permissions):
    - A `Role` table.
    - A `UserRole` table pairing `User` and `Role` entries. This is sometimes called *RBAC* (Role-based Access Control).

Middleware

FastAPI enables insertion of code at the web layer that:

- Intercepts the request
- Does something with the request
- Passes the request to a path function
- Intercepts the response returned by the patch function
- Does something with the response
- Returns the response to the caller

It’s similar to what a Python decorator does to the function that it “wraps”.

In some cases, you could use wither middleware or dependency injection with `Depends()`. Middleware is handier for more global security issues like CORS, which brings up …

CORS

CORS (Cross-Origin Resource Sharing) involves communication between other trusted servers and your website. If your site has all the frontend and backend code in one place, then there’s no problem. But these days, it’s very common to have a JavaScript frontend talking to a backend written in something like FastAPI. These servers will not have the same *origin*:

- Protocol: `http` or `https`
- Domain: Internet domain, like `google.com` or `localhost`

- Port: Numeric TCP/IP port on that domain, like `80`, `443`, or `8000`.

How does the backend know a trustable frontend from a box of moldy radishes, or a mustache-twirling attacker? That's a job for CORS, which specifies what the backend trusts, the most prominent being:

- Origins
- HTTP methods
- HTTP headers
- CORS cache timeout

You hook into CORS at the web level. Example 11-X shows how to allow only one frontend server, (with the domain *https://ui.cryptids.com*) and any HTTP headers or methods:

Example 11-13. Activate CORS middleware

```
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://ui.cryptids.com",],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
    )

@app.get("/test_cors")
def test_cors(request: Request):
    print(request)
```

Once that's done, any other domain that tries to contact your backend site directly will be refused.

Third Party Packages

You've now read some examples of how to code authentication and authorization solutions with FastAPI. But maybe you don't need to do everything yourself. The FastAPI ecosystem is growing fast, and there may be packages available that do a lot of the work for you.

Here are some untested examples. There are no guarantees that any package in this list will still be around and supported over time, but they may be worth a look:

- fastapi-users
- fastapi-jwt-auth
- fastapi-login
- fastapi-auth0
- authx
- fastapi-user-auth
- fastapi-authz
- fastapi-opa
- fastapi-key-auth
- fastapi-auth-middleware
- fastapi-jwt
- fastapi_auth2
- fastapi-sso
- fief

Review

This was a heavier chapter than most. It showed some ways that you can authenticate visitors, and authorize them to do certain things. These are two aspects of web security, followed by some discussion of CORS.

---

[1] If I were paid by the line, fate might have intervened.

Chapter 12. Testing

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

—Brenan Keller, Twitter

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 12th 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

This chapter discusses the kinds of testing that you would perform on a FastAPI site: *unit*, *integration*, and *full*. It features *pytest* and automated test development.

Web API Testing

You've already seen a number of manual API testing tools as endpoints have been added:

- Httpie
- Requests

- Httpx
- The web browser

And there are many more:

- Curl is very well known, although in this book I’ve used Httpie instead for its simpler syntax.
- Httpbin, written by the author of Requests, is a free test server that provides many views into your HTTP request.
- Postman is a full API test platform.
- Chrome Developer Tools is a rich toolset, part of the Chrome browser.

These can all be used for full (end-to-end) tests, such as those you’ve seen in the previous chapters. Those manual tests have been useful for quickly verifying code just after it’s typed.

But what if some change that you make later breaks one of those earlier manual tests (a *regression*)? You don’t want to rerun dozens of tests after every code change. That’s when *automated* tests become important.

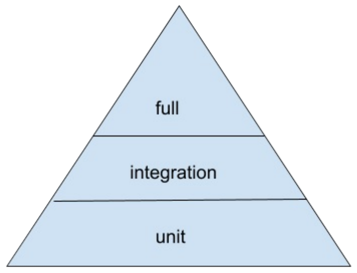
The rest of this chapter focuses on these, and how to build them with Pytest.

Where to Test

I’ve mentioned the different varieties of tests:

- *Unit*: Within a layer, tests individual functions.
- *Integration*: Across layers, tests connectivity.
- *Full*: Tests the full API and stack beneath it.

Sometimes these are called a *test pyramid*, with the width indicating relatively how many tests should be in each group:



What to Test

What should you test as you’re writing code? Basically, for a given input, confirm that you get the correct output. You might check:

- Missing inputs.
- Duplicate inputs.
- Incorrect input types.
- Incorrect input order.
- Invalid input values.
- Huge inputs or outputs.

Errors can happen anywhere:

- *The Web layer*: Pydantic will catch any mismatch with the model and return a 422 HTTP status code.
- *The Data layer*: The database will raise exceptions for missing or duplicate data, as well as SQL query syntax errors. Timeouts or memory exhaustion may occur when passing a huge data result in one piece, instead of in chunks with a generator or pagination.
- *Any layer*: Plain old bugs and oversights.

Chapters 8, 9, and 10 contained some:

- *Full manual tests*: Using tools like Httpie
- *Unit manual tests*: As Python fragments

- *Automated tests*: Using Pytest scripts

The next few sections will expand on Pytest.

Pytest

Python has long had the standard package unittest. A later third-party package called nose tried to improve on it. Most Python developers now prefer Pytest, which does more than either of these and is easier to use. It isn't built into Python, so you'll need to run `pip install pytest` if you don't already have it. Also, run `pip install pytest-mock` to get the automatic `mocker` fixture; you'll see this later in this chapter.

What does Pytest offer? Nice automatic things include:

- *Test discovery*: A *test prefix or test* suffix in a Python file name will be run automatically. This goes down into subdirectories, executing as many tests as you have there.
- *Assertion failure details*: A failing `assert` statement prints what was expected and what actually happened.
- *Fixtures*: Functions that can run once for the whole test script, or run for every test (its *scope*), providing test functions with parameters like standard test data or database initialization. Fixtures are a sort of dependency injection, like FastAPI offers for web path functions: specific data passed to a general test function.
- *Parameterization*: Provides multiple test data to a test function.

Layout

Where should you put your tests? There doesn't seem to be wide agreement, but two reasonable designs are:

- A *test* directory at the top, with subdirectories for the code area being tested (like *web*, *service*, etc.)

- A *test* directory under each code directory (like *web*, *service*, etc.).

Also, within the specific subdirectory like *test/web*, should you make more directories for different test types (like *unit*, *integration*, and *full*)? In this book, I’m using this hierarchy:

- *test*
    - *unit*
        - *web*
        - *service*
        - *data*
    - *integration*
    - *full*

Individual test scripts live within the bottom directories. Those are in this chapter.

Automated Unit Tests

A unit test should check one thing, within one layer. This usually means passing some parameter(s) to a function and asserting what should be returned.

Unit tests requires *isolation* of the code being tested. If not, you’re also testing something else. So, how do you isolate code for unit tests?

Mocking

In this book’s code stack, accessing a URL via a web API generally calls a function in the Web layer, which calls a function in the Service layer, which calls a function in the Data layer, which accesses a database. The results flow back up the chain, eventually back out of the Web layer to the caller.

Unit testing sounds simple. For each function in your code base, pass in some test arguments and confirm that it returns expected values. Thie works well for a *pure function*: one that takes input arguments and returns responses without referencing any external code. But most functions also call other functions, so how can you control what those other functions do? What about the data that come from these external sources? The most common external thing to control is database access, but really it can be anything.

One method is to *mock* each external function call. Because functions are first-class objects in Python, you can substitute one function for another. The `unittest` package has a `mock` module that does this.

Many developers believe that mocking is the bext way to isolate unit tests. I'll first show examples of mocking here, along with the argument that often mocking requires too much knowledge of *how* your code works, rather than the results. You may hear the terms *structural testing* (as in mocks, where the tested code is quite visible) and *behavioral testing* (where the code internals are not needed).

Example 12-1 and 12-2 define the modules *mod1.py* and *mod2.py*.

Example 12-1. Called module (mod1.py)

```
def preamble() -> str:
    return "The sum is "
```

Example 12-2. Calling module (mod2.py)

```
import mod1

def summer(x: int, y:int) -> str:
    return mod1.preamble() + f"{x+y}"
```

The `summer()` function calculates the sum of its arguments, and returns a string with a preamble and the sum. Example 12-3 is a very minimal pytest script to verify `summer()`.

Example 12-3. Pytest script test_summer1.py

```
import mod2

def test_summer():
    assert "The sum is 11" == mod2.summer(5,6)
```

Example 12-4 runs it successfully.

Example 12-4. Run Pytest script

```
$ pytest -q test_summer1.py
.
```

```
[100%]
1 passed in 0.04s
```

(The `-q` runs the test quietly, without lots of extra printed details.) Okay, it passed. But the `summer()` function got some text from the `preamble` function. What if we just want to test that the addition succeeded?

We could write a new function that just returns the stringized sum of two numbers, then rewrite `summer()` to return this appended to the `preamble()` string.

Or, we could mock `preamble()` to remove its effect, as shown in multiple ways in Example 12-5.

Example 12-5. Pytest with a mock (test_summer2.py)

```
from unittest import mock
import mod1
import mod2

def test_summer_a():
    with mock.patch("mod1.preamble", return_value=""):
        assert "11" == mod2.summer(5,6)

def test_summer_b():
    with mock.patch("mod1.preamble") as mock_preamble:
        mock_preamble.return_value=""
        assert "11" == mod2.summer(5,6)

@mock.patch("mod1.preamble", return_value="")
def test_summer_c(mock_preamble):
    assert "11" == mod2.summer(5,6)

@mock.patch("mod1.preamble")
def test_caller_d(mock_preamble):
    mock_preamble.return_value = ""
    assert "11" == mod2.summer(5,6)
```

These tests show that mocks can be created in more than one way. Function `test_caller_a()` uses `mock.patch()` as a Python *context manager* (the `with` statement). Its arguments are:

- `"mod1.preamble"`: the full string name of the `preamble()` function in module `mod1`.
- `return_value=""` makes this mocked version return an empty string.

Function `test_caller_b()` is almost the same, but adds `as mock_preamble` to use the mock object on the next line.

Function `test_caller_c()` defines the mock with a Python *decorator*. The mocked object is passed as an argument to `test_caller2()`.

Function `test_caller_d()` is like `test_caller_b()`, setting the `return_value` in a separate call to `mock_preamble`.

In each case, the string name of the thing to be mocked must match how it's called in the code that's being tested (in this case, `summer()`). The mock library converts this string name to a variable that will intercept any references to the original variable with that name. (Remember that in Python, variables are just references to the real objects.)

So, when Example 12-6 below is run, in all four `summer()` test functions, when `summer(5,6)` is called, the changeling mock `preamble()` is called instead of the real one. The mocked version drops that string, so the test can ensure that `summer()` returns a string version of the sum of its two arguments.

Example 12-6. Run mocked Pytest

```
$ pytest -q test_summer2.py
....
[100%]
4 passed in 0.13s
```

NOTE

That was a contrived case, for simplicity. Mocking can be quite complex; see articles like this for clear examples, and the official docs for the harrowing details.

Test Doubles and Fakes

To perform that mock, you needed to know that the `summer()` function imported the function `preamble()` from the module `mod1`. This was a structural test, requiring knowledge of specific variable and module names.

Is there a way to perform a behavioral test that doesn't need this?

One way is to define a *double*: separate code that does what we want in the test (in this case, make `preamble()` return an empty string). One way to do this is with imports. I'll apply this to this example first, before using it for unit tests in the layers of the next three sections.

First, redefine *mod2.py* in Example 12-7.

Example 12-7. Make mod2.py import a double if unit testing

```
import os
if os.get_env("UNIT_TEST"):
    import fake_mod1 as mod1
else:
    import mod1

def summer(x: int, y:int) -> str:
    return mod1.preamble() + f"{x+y}"
```

Example 12-8 defines that double module *fake_mod1.py*.

Example 12-8. Double fake_mod1.py

```
def preamble() -> str:
    return ""
```

And Example 12-9 is the test.

Example 12-9. Test script test_summer_fake.py

```
import os
os.environ["UNIT_TEST"] = "true"
import mod2

def test_summer_fake():
    assert "11" == mod2.summer(5,6)
```

….which Example 12-10 runs.

Example 12-10. Run the new unit test

```
$ pytest -q test_summer_fake.py
.
[100%]
1 passed in 0.04s
```

This import-switching method does require adding a check for an environment variable, but avoids having to write specific mocks for function calls. You can be the judge of which you prefer. In the next few sections, I'll use the `import` method, which works nicely with the *fake* package that I'd been using as I defined the code layers.

Summarizing: these examples replaced `preamble()` with a *mock* in a test script, or imported a doppelgänger *double*. There are other ways to isolate the code being tested, but these work, and are not as tricky as others that Google might find for you.

Web

This layer implements the site's API. Ideally, each path function (endpoint) should have at least one test — maybe more, if the function could fail in more than one way. At the Web layer, you normally want to see if the endpoint exists, works with the correct parameters, and returns the right status code and data.

NOTE

These are shallow API tests, testing solely within the Web layer. So, Service layer calls (which would in turn call the Data layer and the database) need to be intercepted, along with any other calls that exit the Web layer.

Using the `import` idea of the previous section, I'll use the environment variable `CRYPTID_UNIT_TEST` to import the *fake* package as `service`, instead of the real `service`. This stops Web functions from calling Service functions, and insteads short-circuits them to the *fake* (doubles) version. Then the lower Data layer and database aren't involved, either. We get what we want: unit tests. Example 12-11 has the modified *web/creature.py* file.

Example 12-11. Modified web/creature.py

```
import os
from fastapi import APIRouter, HTTPException
from model.creature import Creature
if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import creature as service
else:
    from service import creature as service
from error import Missing, Duplicate

router = APIRouter(prefix = "/creature")

@router.get("/")
def get_all() -> list[Creature]:
    return service.get_all()
```

```
@router.get("/{name}")
def get_one(name) -> Creature:
    try:
        return service.get_one(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.post("/", status_code=201)
def create(creature: Creature) -> Creature:
    try:
        return service.create(creature)
    except Duplicate as exc:
        raise HTTPException(status_code=409, detail=exc.msg)

@router.patch("/")
def modify(name: str, creature: Creature) -> Creature:
    try:
        return service.modify(name, creature)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)


@router.delete("/{name}")
def delete(name: str) -> None:
    try:
        return service.delete(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)
```

Example 12-12 has tests, using two Pytest *fixtures*:

- `sample()`: a new `Creature` object
- `fakes()`: a list of “existing” creatures

The fakes are obtained from a lower level module. By setting the environment variable `CRYPTID_UNIT_TEST`, the Web module in Example 12-11 imports the fake service version (providing fake data rather that calling the database) rather than the real one. This isolates the tests, which was the point.

Example 12-12. Web unit tests for creatures, using fixtures

```
from fastapi import HTTPException
import pytest
import os
os.environ["CRYPTID_UNIT_TEST"] = "true"
from model.creature import Creature
from web import creature
from error import Missing, Duplicate

@pytest.fixture
def sample() -> Creature:
    return Creature(name="dragon",
        description="Wings! Fire! Aieee!",
        location="worldwide")

@pytest.fixture
def fakes() -> list[Creature]:
    return creature.get_all()

def assert_duplicate(exc):
    assert exc.value.status_code == 404
    assert "Duplicate" in exc.value.msg

def assert_missing(exc):
    assert exc.value.status_code == 404
    assert "Missing" in exc.value.msg

def test_create(sample):
    assert creature.create(sample) == sample

def test_create_duplicate(fakes):
    with pytest.raises(HTTPException) as exc:
        resp = creature.create(fakes[0])
        assert_duplicate(exc)
        def test_get_one(fakes):
    assert creature.get_one(fakes[0].name) == fakes[0]

def test_get_one_missing():
    with pytest.raises(HTTPException) as exc:
        resp = creature.get_one("bobcat")
        assert_missing(exc)

def test_modify(fakes):
    assert creature.modify(fakes[0].name, fakes[0]) == fakes[0]

def test_modify_missing(sample):
    with pytest.raises(HTTPException) as exc:
        resp = creature.modify(sample.name, sample)
        assert_missing(exc)

def test_delete(fakes):
    assert creature.delete(fakes[0].name) is None

def test_delete_missing(sample):
    with pytest.raises(HTTPException) as exc:
        resp = creature.delete("emu")
        assert_missing(exc)
```

Service

In a way, this is the important layer, and could be connected to different Web and Data layers. Example 12-13 is very similar to Example 12-11, differing mainly in the `import` and use of the lower level `data` module. It also doesn't catch any exceptions that might arise fron the Data layer, leaving them to be handled by the Web layer.

Example 12-13. Modified service/creature.py

```
import os
from model.creature import Creature
if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import creature as data
else:
    from data import creature as data

def get_all() -> list[Creature]:
    return data.get_all()

def get_one(name) -> Creature:
    return data.get_one(name)

def create(creature: Creature) -> Creature:
    return data.create(creature)

def modify(name: str, creature: Creature) -> Creature:
    return data.modify(name, creature)

def delete(name: str) -> None:
    return data.delete(name)
```

Example 12-14 has the corresponding unit tests.

Example 12-14. Service tests in test/unit/service/test_creature.py

```
import os
os.environ["CRYPTID_UNIT_TEST"]= "true"
import pytest

from model.creature import Creature
from error import Missing, Duplicate

@pytest.fixture
def sample() -> Creature:
    return Creature(name="yeti",
        description="Abominable Snowman",
        location="Himalayas")

def test_create(sample):
    resp = data.create(sample)
    assert resp == sample

def test_create_duplicate(data):
    resp = data.create(data)
    assert resp == data
    with pytest.raises(Duplicate):
        resp = service.create(data)

def test_get_exists(data):
    resp = data.create(data)
    assert resp == data
    resp = data.get_one(data.name)
    assert resp == data

def test_get_missing():
    with pytest.raises(Missing):
        resp = data.get_one("boxturtle")

def test_modify(data):
    data.location = "Sesame Street"
    resp = data.modify(data.name, data)
    assert resp == data

def test_modify_missing():
    bob: Creature = Creature(name="bob",
        description="some guy", location="somewhere")
    with pytest.raises(Missing):
        resp = data.modify(bob.name, bob)
```

Data

This layer is simpler to test in isolation, because there’s no worry about accidentally calling some function in an even lower layer. Unit tests should cover both the functions in this layer, and the specific database queries that they use. So far, SQLite has been the database “server” and SQL the query language. But as I mention in Chapter 17, you may decide to work with a package like SQLALchemy, and use its SQL Expression Language or its ORM. Then these would need full tests. So far, I’ve kept to the lowest level — Python’s DB-API and vanilla SQL queries.

Unline the Web and Service unit tests, this time we don’t need “fake” modules to replace the existing Data layer modules. Instead, set a different environment variable to get the Data layer to use a memory-only SQLite instance, instead of a file-based one. This doesn't require any changes to the existing Data modules, just a setting in Example 12-15 test *before* importing any Data modules.

Example 12-15. Data unit tests test/unit/data/test_creature.py

```
import os
import pytest
from model.creature import Creature
from error import Missing, Duplicate

# set this before data.init import below
os.environ["CRYPTID_SQLITE_DB"] = ":memory:"
from data import init, creature

@pytest.fixture
def sample() -> Creature:
    return Creature(name="yeti",
        description="Abominable Snowman",
        location="Himalayas")

def test_create(sample):
    resp = creature.create(sample)
    assert resp == sample

def test_create_duplicate(sample):
    with pytest.raises(Duplicate):
        resp = creature.create(sample)

def test_get_one(sample):
    resp = creature.get_one(sample.name)
    assert resp == sample

def test_get_one_missing():
    with pytest.raises(Missing):
        resp = creature.get_one("boxturtle")

def test_modify(sample):
    creature.location = "Sesame Street"
    resp = creature.modify(sample.name, sample)
    assert resp == sample

def test_modify_missing():
    thing: Creature = Creature(name="snurfle",
        description="some thing", location="somewhere")
    with pytest.raises(Missing):
        resp = creature.modify(thing.name, thing)
```

```
def test_delete(sample):
    resp = creature.delete(sample.name)
    assert resp is None

def test_delete_missing(sample):
    with pytest.raises(Missing):
        resp = creature.delete(sample.name)
```

Automated Integration Tests

Integration tests see how well different code interacts *between* layers. But if you look for examples of this, you get many different answers. Should you test partial call trails like Web → Service, Web → Data, and so on?

To fully test every connection in an A → B → C pipeline, you'd need to test:

- A → B
- B → C
- A → C

And the arrows would fill a quiver if you have more than these three junctions.

Or should integration tests be essentially full tests, but with the very end piece - data storage on disk — mocked?

Sa far, I've been using SQLite as the database, and we can use in-memory SQLite as a double (fake) for the on-disk SQLite database. If your queries are *very* standard SQL, SQLite-in-memory may be an adequate mock for other databases as well. If not, there are modules that are tailored to mock specific databases:

- *PostgreSQL*: pgmock
- *MongoDB*: mongomock
- *Many*: python-mock-resources spins up various test databases in Docker containers, and is integreted with Pytest.

Finally, you could just fire up a test database of the same kind as production. An environment variable could contain the specifics, much like the unit test / fake trick I’ve been using.

The Repository Pattern

Although I did not implement it for this book, the repository pattern is an interesting approach. A *repository* is a simple intermediate in-memory data store — like the fake Data layer that you’ve seen here so far. This then talks to pluggable backends for real databases. It’s accompanied by a *unit of work* pattern which ensures that a group of operations in a single *session* is either committed or rolled back as a whole. So far, the database queries in this book have been atomic. For real-world database work, you may need multistep queries, and some kind of session handling.

The repository pattern also dovetails with dependency injection, which you’ve seen elsewhere here and probably appreciate a little by now.

Automated Full Tests

Full tests exercise all the layers together, as close to production use as possible. Most of the tests that you’ve already seen in this book have been full — call the Web endpoint, run through Servicetown to downtown Dataville, and return with groceries. These are blackbox tests. Everything is live, and you don’t care how it does it, just that it does it.

You can fully test each endpoint in the overall API in two ways:

- *Over HTTP/HTTPS*: Write individual Python test clients that access the server. Many examples in this book have done this, with standalong clients like Httpie, or in scripts using Requests.
- *Using `TestClient`*: Use this built-in FastAPI/Starlette object to access the server directly, without an overt TCP connection.

But this requires writing one or more tests for each endpoint. This can become medieval, and we’re a few centuries past medieval now. A more recent approach is based on *property-based testing*. This takes advantage of FastAPI’s autogenerated documentation. An OpenAPI *schema* called *openapi.json* is created by FastAPI every time you change a path function or path decorator in the Web layer. This schema details everything about every endpoint: arguments, return values, and so on. That's what OpenAPI is for:

> *The OAS defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic.*
>
> —https://www.openapis.org/faq

Two packages are needed:

- Hypothesis: `pip install hypothesis`
- Schemathesis: `pip install schemathesis`

Hypothesis is the base library, and Schemathesis applies it to the OpenAPI 3.0 schema that FastAPI generates. Running Schemathesis reads this schema, generates gobs of tests with varying data (that you don't need to come up with!), and works with Pytest.

To keep this brief, Example 12-16 first slims *main.py* down to its base creature and explorer endpoints:

Example 12-16. Bare-bones main.py

```
from fastapi import FastAPI
from web import explorer, creature

app = FastAPI()
app.include_router(explorer.router)
app.include_router(creature.router)
```

Exmople 12-17 runs the tests.

Example 12-17. Run schemathesis tests

```
$ schemathesis http://localhost:8000/openapi.json
===================== Schemathesis test session starts
=====================
Schema location: http://localhost:8000/openapi.json
Base URL: http://localhost:8000/
Specification version: Open API 3.0.2
Workers: 1
Collected API operations: 12
```

```
GET /explorer/ .                                                      [
8%]
POST /explorer/ .                                                     [
16%]
PATCH /explorer/ F                                                    [
25%]
GET /explorer .                                                       [
33%]
POST /explorer .                                                      [
41%]
GET /explorer/{name} .                                                [
50%]
DELETE /explorer/{name} .                                             [
58%]
GET /creature/ .                                                      [
66%]
POST /creature/ .                                                     [
75%]
PATCH /creature/ F                                                    [
83%]
GET /creature/{name} .                                                [
91%]
DELETE /creature/{name} .
[100%]
```

I got two F’s, both in `PATCH` calls (`modify()` functions). How mortifying.

This output section is followed by one marked `FAILURES`, with detailed stack traces of any tests that failed. Those need to be fixed. The final section is marked `SUMMARY`:

```
  Performed checks:
      not_a_server_error                    717 / 727 passed
  FAILED

  Hint: You can visualize test results in Schemathesis.io
  by using `--report` in your CLI command.
```

That was fast, and I didn’t have to create multiple tests for each endpoint, imagining inputs that might break them. Property-based testing reads the types and constraints of the input arguments from the API schema, and generates a range of values to shoot at each endpoint.

This is yet another unexpected benefit of type hints, which at first seemed to be just nice things:

type hints → OpenAPI schema → generated documentation *and* tests

Security Testing

Security isn't one thing, but everything. You need to defend against malice, but also against plain old mistakes, and even events that you have no control over. Let's defer scaling issues to the next section, and deal mainly here with the analysis of potential threats.

Chapter 11 discussed authentication and authorization. These factors are always messy and error-prone. It's tempting to use clever methods to counteract clever attacks, and it's always a challenge to design protection that's easy to understand and implement.

But now that we know about Schemathesis, read its documentation on property-based testing for authentication. Just as it vastly simplified testing most of the API, it can automate much of the tests for endpoints that need authentication.

(MORE: authorization details?)

Load Testing

Load tests exercise how your application handles heavy traffic:

- API calls
- Database reads or writes
- Memory use
- Disk use
- Network latency and bandwidth

Some can be *full* tests that simulate an army of users clamoring to use your service; you want to be ready before that day arrives. There's some overlap between this section and Chapter 14 (Performance) and Chapter 15 (Troubleshooting).

There are many good load testers out there, but here I'll use one called Locust.

With Locust, you define all your tests with plain Python scripts. It can simulate hundreds of thousands of users, all pounding away at your site, or even multiple servers, at once.

Install it locally with `pip install locust`.

The first thing you may want to test is how well your site stands up. How many concurrent visitors can your site handle? This is like testing how much extreme weather a building can withstand when faced with a hurricane/earthquake/blizzard, or other home insurance event. So, you need some website structural tests.

Example 12-18 aims the Locust firehose at the current creature/explorer site.

Example 12-18. Run locust load tests

```
$ ...
----
```

But there’s more! Recently, Grasshopper extended Locust to do things like measuring time across multiple HTTP calls. To try it out, install with `pip install locust-grasshopper`.

Review

This chapter fleshed out the types of testing, with examples of Pytest performing automated code testing at the unit, integration, and full levels. API tests can be automated with schemathesis. It also discussed how to expose security and performance problems before they strike.

Chapter 13. Production

If builders built buildings the way programmers wrote programs, the first woodpecker that came along would destroy civilization.

—Gerald Weinberg

Preview

You have an application running on your local machine, and now you’d like to share it. This chapter shares many scenarios on how to move it to production, and keep it running correctly and efficiently. Because some of the details can be *very* detailed, in some cases I’ll refer to helpful external documents rather than than stuffing them in here.

Deployment

In all of the code examples in this book so far, I’ve used a single instance of `uvicorn` running on `localhost`, port `8000`. To handle lots of traffic, you want multiple servers, running on the multiple cores that modern hardware provides. You’ll also need something above these servers to:

- Keep them running (a *supervisor*)
- Gather and feed external requests (a *reverse proxy*)
- Return responses
- Provide HTTPS “termination”

Multiple Workers

You’ve probably seen another Python server called gunicorn. This can supervise multiple workers, but it’s a WSGI server, and FastAPI is based on ASGI. Luckily, there’s a special uvicorn worker class that can be managed by gunicorn.

Example 13-X sets up these uvicorn workers on localhost, port 8000 (this is adapted from the official documentation). The quotes protect the shell from any special interpretation.

Example 13-1. Use gunicorn with uvicorn workers

```
$ pip install "uvicorn[standard]" gunicorn
$ gunicorn main:app --workers 4 --worker-class \
uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000
```

You’ll see many lines as it does your bidding. It will start a top-level gunicorn process, talking to four uvicorn worker sub-processes, all sharing port `8000` on `localhost` (`0.0.0.0`). Change the host, port, or number of workers if you want something else. The `main:app` refers to *main.py*, and the FastAPI object with the variable name `app`. The gunicorn docs claim:

> Gunicorn should only need 4-12 worker processes to handle hundreds or thousands of requests per second.

It turns out that uvicorn itself can also fire up multiple uvicorn workers:

Example 13-2. Use uvicorn with uvicorn workers

```
$ uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4
```

But this method doesn’t do process management, so the gunicorn method is usually preferred. There are other process managers for uvicorn: see its official docs.

This handles three of the four jobs mentioned in the previous section, but not

HTTPS encryption.

HTTPS

The official FastAPI HTTPS docs, like all of the official FastAPI docs, are extremely informative. I recommend reading this, followed by Sebastián’s description of how to add HTTPS support to FastAPI by using Traefik. Traefix sits “above” your web servers, similar to nginx as a reverse proxy and load balancer, but it includes that HTTPS magic.

Although there are many steps in the process, it’s still much simpler than it used to be. In particular, you used to regularly pay big bucks to a Certificate Authority for a digital certificate that you could use to provide HTTPS for your site. Luckily, those authorities been largely replaced by the free service Let’s Encrypt.

Docker

When Docker burst on the scene (in a five minute lightning talk by Solomon Hykes of dotCloud at PyCon 2013), it was the first time most of us had ever heard of Linux containers. Over time, we learned that Docker was faster and lighter than virtual machines. Instead of emulating a full operating system, each container shared the server’s Linux kernel, and isolated processes and networks into their own namespaces. Suddenly, by using the free Docker software, you could host multiple independent services on a single machine, without worrying about them stepping all over one another.

Ten years later, Docker is universally recognized and supported. If you want to host your FastAPI application on some cloud service, you’ll usually need to create a *Docker image* of it first. The official FastAPI docs include a thorough description of how to build a Dockerized version of your FastAPI application. One step is to write a *Dockerfile*: a text file containing some Docker configuration info, like what application code to use and what processes to run. Just to prove that this isn’t brain surgery during a rocket launch, here’s the Dockerfile from that page:

```
FROM python:3.9
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
```

```
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
COPY ./app /code/app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

I recommend reading the official docs, or other links that a Google search of `fastapi docker` will produce.

Cloud Services

Many sources of paid or free hosting are available on the Net. Some walkthoughs on how to host FastAPI with them include:

- Deta
- Linode
- Heroku

Kubernetes

Kubernetes grew from some internal Google code for managing internal systems that were becoming ever more godawfully complex. System administrators (as they were called then) used to manually configure tools like load balancers, reverse proxies, humidors[1], and so on. Kubernetes aimed to take much of this knowledge and automate it: don't tell me *how* to handle this, tell me what you *want*. This included tasks like keeping a service running, or firing up more servers if traffic spikes.

There are many descriptions of how to deploy FastAPI on Kubernetes, including this brief outline.

Performance

FastAPI's performance is currently among the highest of any Python web framework, even comparable to frameworks in faster languages like Golang. But much of this is due to ASGI, avoiding I/O waiting with async. Python itself is a relatively slow language. Ths following are some tips and tricks to improve overall performance.

Async

Often, a web server doesn't need to be really fast. It spends much of its time getting HTTP network requests and returning results (the Web layer in this book). In between, a web service performs business logic (the Service layer) and accesses data sources (the Data layer), and again spends much of its time on network I/O.

Whenever code in the web service has to wait for a response, it's a good candidate to use an async function (`async def` rather than `def`). This lets FastAPI and Starlette schedule the async function and do other things while waiting for it to get its response. This is one of the reasons why FastAPI's benchmarks are better than WSGI-based frameworks like Flask and Django.

Performance has different aspects:

- The time to handle a single request
- The number of requests that can be handled at once

Caches

If you have a web endpoint that ultimately gets data from a static source (like a database record that changes rarely or never), it's possible to *cache* the data in a function. This could be in any of the layers. Python provides the standard functools module and the functions `cache()` and `lru_cache()`.

Databases, Files, and Memory

One of the most common causes of a slow website is a missing index for a database table of sufficient size. Often, you won't see the problem until your table has grown to particular size, and then queries suddenly become much slower. In SQL, any column in a `WHERE` cluse should be indexed.

In many examples in this book, the primary key of the `creature` and `explorer` tables has been the text field `name`. When the tables were created, `name` was declared the `primary key`. For the tiny tables that you've seen so far in this book, SQLite would just ignore that key anyhow, since it's faster just to scan the table. But once a table gets to a decent size — say a million rows — a missing index will make a noticeable difference. The solution: run a query optimizer.

Even if you have a small table, you can do some database *load testing* with Python scripts or open-source tools.

If you’re making a number of sequential database queries, it may be possible to combine them in a single *batch*.

If you’re uploading or downloading a large file, use the streaming versions rather than a giant gulp.

Queues

If you’re performing any task that takes longer than a fraction of a second (like sending a confirmation email or downsizing an image), it may be worth handing it off to a job queue like Celery.

Python Itself

If your web service seems slow because it does significant computing with Python, you may want a “faster Python”. Alternatives include:

- Use Pypy instead of the standard CPython.
- Write a Python extension in C, C++, or Rust.
- Convert the slow Python code to Cython (used by Pydantic and Uvicorn themselves).

A very intriguing recent announcement was the Mojo language. It aims to be a complete superset of Python, with new features (with the same friendly Python syntax) that can speed up a Python example by *thousands* of times. The main author, Chris Lattner, had previously worked on compiler tools like LLVM, Clang, and MLIR, plus the Swift language at Apple.

Mojo aims to be a single-language soution to AI development, which now (in Pytorch and TensorFlow) requires Python/C/C++ sandwiches that are hard to develop, manage and debug. But Mojo also would be a good general-purpose language aside from AI.

I coded in C for years, and kept waiting for a successor that was as performant but as easy to use as Python. D, Golang, Julia, Zig, and Rust were possibilities, but if Mojo can live up to its goals, I would use Mojo extensively.

Troubleshooting

Look bottom-up from the time and place where you encounter a problem. This includes time and space performance issues, but also logic and async traps.

Kinds of Problems

At a first glance, what HTTP response code did you get?

- `404`: An authentication or authorization error.
- `422`: Usually a Pydantic complaint about use of a model.
- `500`: Some service behind your FastAPI one failed.

Logging

Uvicorn and other web servers normally write logs to stdout. You can check the log to see what call was actually made, including the HTTP verb and URL, but not data in the body, headers, or cookies.

If a particular endpoint returned a 400-level status code, you can try feeding the same input back and see if the error reoccurs. If so, my first caveman debugging instinct is to add `print()` statements in the relevant Web, Service, and data funcions.

Also, wherever you raise an exception, add details. If a database loookup fails. include the input values and specific error, like an attempt to double a unique key field.

Metrics

The terms *metrics*, *monitoring*, *observability*, and *telemetry* may seem to overlap. It’s common practice in Pythonland to use:

- Prometheus to gather metrics.
- Grafana to display them.
- OpenTelemetry to measure timing.

You can apply these to all of your site's layers: Web, Service, and Data. The Service ones may be more business-ortented, and the others more technical, and useful for site developers and maintainers.

Some links to gather FastAPI metrics:

- Prometheus FastAPI Instrumentator
- Getting Started: Monitoring a FastAPI App with Grafana and Prometheus - A Step-by-Step Guide
- FastAPI Observability
- OpenTelemetry FastAPI Instrumentation
- OpenTelemetry FastAPI Tutorial - complete implementation guide
- A language-specific implementation of OpenTelemetry in Python

Review

It's pretty clear that production is not easy. Problems include the web machinery itself, network and disk overloading, and database problems. This chapter offered some hints on how to get the information you need, and where to start digging when problems pop up.

---

[1] Wait, that keeps cigars fresh.

Part IV. A Gallery

Part III built a minimal website with some basic code. Now let’s do something fun with it. The following chapters apply FastAPI to common web uses — forms, files, databases, charts and graphics, maps, and games.

To tie these applications together, and make them more interesting than the usual dry computing book examples, we’ll plunder data from an unusual source, some of which you’ve already glimpsed — imaginary creatures from world folkore, and the explorers who pursue them. There will be yetis, but also more obscure — though no less striking — members.

Chapter 14. Databases, Data Science, and a Little AI

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 14th 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

This chapter discusses how to use FastAPI to store and retrieve data. It expands on the simple SQLite examples of chapter 10 with:

- Other open source databases (relational and not).
- Higher-level uses of SQLAlchemy.
- Better error checking.

Data Storage Alternatives

NOTE

The term *database* is unfortunately used to refer to three different things:

- The server *type*, like PostgreSQL, SQLite, or MySQL.

- A running instance of that *server*.
- A *collection of tables* on that server.

To avoid confusion, like referring to an instance of the last one above as a “PostgreSQL database database database”, I’ll attach other terms to indicate which one I mean.

The usual backend for a website is a database. They’re peanut butter and jelly, and although you could conceivably store your data in other ways (or pair peanut butter with pickles), for this book we’ll stick with databases.

Databases handle many problems that you would otherwise have to solve yourself with code, such as:

- Multiple access
- Indexing
- Data consistency

The general choices are:

- Relational databases, with the SQL query language.
- Non-relational databases, with various query languages.

Relational Databases and SQL

Python has a standard relational API definition called DB-API, and it’s supported by Python driver packages for all the major databases.

Database	Python Drivers
Open source	
SQLite	sqlite3
PostgreSQL	psycopg2, asyncpg

MySQL	MySQLDB, PyMySQL
Commercial	
Oracle	oracledb
SQL Server	pyodbc, pymssql
DB/2	ibm_db

The main Python packages for relational databases and SQL are:

- SQLAlchemy: A very full-featured library that can be used at many levels.
- SQLModel: A combination of SQLAlchemy and Pydantic, by the author of FastAPI.
- Records: From the author of the requests package, a simple query API.

SQLAlchemy

The most popular Python SQL package is SQLAlchemy. Although many explanations of SQLAlchemy only discuss its ORM (object-reational mapper), it has multiple layers, and I'll discuss these bottom-up.

Core

The base of SQLAlchemy is called *Core*, and it comprises:

- An `Engine` object that implements the DB-API standard.
- URLS that express the SQL server type and driver, and the specific database collection on that server.
- Client-server connection pools.
- Transactions (`COMMIT` and `ROLLBACK`).
- SQL *dialect* differences among different database types.

- Direct SQL (text string) queries.
- Queries in the SQL Expression Language.

Some of these features, like the dialect handling, make SQLAlchemy the package of choice for working with different server types. You can use it to execute plain DB-API SQL statements, or use the SQL Expression Language.

I’ve been using the raw DB-API SQLite driver so far, and will continue. But for larger sites, or those that might need to take advantage of some special server feature, SQLAlchemy (using basic DB-API, SQL Expression Language, or the full ORM) is well worth using.

SQL Expression Language

This is *not* the ORM, but another way of expressing queries against relational tables. It maps the underlying storage structures to Python classes like `Table` and `Column`, and operations to Python methods like `select()` and `insert()`. These functions translate to plain SQL strings, and you can access them to see what happened. It’s independent of SQL server types. If you find SQL difficult, this may be worth trying.

Let’s compare a few examples. Example 17-1 shows yhe plain SQL version.

Example 14-1. Straight SQL code for get_one() in data/explorer.py.

```
def get_one(name: str) -> Explorer:
    qry = "select * from explorer where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    return row_to_model(curs.fetchone())
```

Example 17-2 shows the full SQL Expression Language equivalent to set up the database, build the table, and perform the insertion.

Example 14-2. SQLAlchemy SQL Expression Language example for get_one().

```
from sqlalchemy import Metadata, Table, Column, String,
from sqlalchemy import connect, insert, selectr

conn = connect("sqlite:///cryptid.db")
meta = Metadata()
explorer_table = Table(
    "explorer",
    meta,
    Column("name", Text, primary_key=True),
    Column("nationality", Text),
    )
insert(explorer_table).values(name="Beau Buffette",
    nationality="Unirted States")
_(MORE)_
```

For more examples, some alternative documentation is a bit more readable than the official pages.

ORM

An ORM expresses queries in terms of domain data models, not the relational tables and SQL logic at the base of the database machinery. The official documentation goes into all the details. The ORM is much more complex than the SQL expression language. Developers who prefer fully *object-oriented* patterns usually prefer ORMs.

Many books and articles on FastAPI jump right into SQLAlchemy's ORM when they come to the database section. I understand the appeal, but also know that it requires you to learn another abstraction. SQLAlchemy is an excellent package, but if its abstractions doesn't always hold, then you have two problems. The simplest solution may be to just use SQL, and move to the Expression Language or ORM if the SQL gets too hairy.

SQLModel

The author of FastAPI combined aspects of FastAPI, Pydantic, and SQLAlchemy to make SQLModel. It repurposes some development techniques from the web world to relational databases. SQLModel matches SQLAlchemy's ORM with Pydantic's data definition and validation.

SQLite

I introduced SQLite in Chapter 10, using it for the Data layer examples. It's public domain — you can't get more open sourcey than that. SQLite is used in every browser and every smartphone, making it one of the most widely deployed software packages in the world. It's often overlooked when choosing a relational database, but it's possible that multiple SQLite "servers" could support some large services as well as a beefy server like PostgreSQL.

PostgreSQL

In the early days of relational databases, IBM's System R was the pioneer, and offshoots battled for the new market — mainly open source Ingres versus commercial Oracle. Ingres featured a query language named QUEL, and System R had SQL. Although QUEL was considered better than SQL by some, Oracle's adoption of SQL as a standard, plus IBM's influence, helped push Oracle and SQL to success. Years later, Michael Stonebraker returned to migrate Ingres to PostgreSQL. Nowadays, open source developers tend to choose PostgreSQL, although MySQL was very popular a few years ago and is still around.

EdgeDB

Despite the success of SQL over the years, it does have some design flaws that make queries awkward. Unlike the mathematical theory (*relational calculus*, by E. F Codd), that SQL is based on, the SQL language design itself is not *composable*. Mainly, this means that it's hard to nest queries within larger ones, leading to more complex and verbose code.

So, just for fun, I'm throwing in a new relational database here. EdgeDB was written (in Python!) by the author of Python's asyncio. It's described as "Post-SQL" or *graph-relational*. Under the hood, it uses PostgreSQL to handle the tough systemy stuff. Edge's contribution is EdgeQL: a new query language that aims to avoid those sharp SQL edges; it's actually translated to SQL for PostgreSQL to execute. This post handily compares EdgeQL and SQL. The readable illustrated official documentation parallels the book *Dracula*.

Could EdgeQL spread beyond EdgeDB and become an alternative to SQL? Time will tell.

Non-relational (NoSQL) Databases

Biggies in the open-source "NoSQL" or "NewSQL" world include:

Database	Python Drivers
Redis	redis-py

MongoDB	pymongo, motor
Cassandra	cassandra-driver
ElasticSearch	elasticsearch

NoSQL Databases

Sometimes *NoSQL* means literally “no SQL”, but sometimes “not only SQL”. Relational databases enforce structures on data, often visualized as rectangular tables with column fields and data rows, similar to spreadsheets. There are *normal forms* that specify how these relate, such as only allowing a single value per cell (row/column intersection).

NoSQL databases relax these rules, sometimes allowing varying column/field types across individual data rows. Often the *schemas* (database designs) can be ragged structures, like you could express in JSON or Python, rather than relational boxes.

Redis

Redis is a data structure server that runs completely in memory, although it can save to and restore from disk. It closely matches Python’s own data structures, and has become extremely popular.

MongoDB

MongoDB is sort of the PostgreSQL of NoSQL servers. A *collection* is the equivalent of a SQL table, and a *document* is the equivalent of a SQL table row. Another difference, and the main reason for a NoSQL database in the first place, is that you don’t need to define what a document looks like. In other words, there’s no fixed *schema*. A document is like a Python dictionary, with any string as a key.

NoSQL Features in SQL Databases

Relational databases were traditionally *normalized* — constrained to follow different levels of rules called *normal forms*. One basic rule was that the value in each cell (row-column intersection) had to be a *scalar* (no arrays or other structures).

NoSQL (or *document*) databases supported JSON directly, and were usually your only choice if you had “uneven” or “ragged” data structures. They were often *denormalized*: all the data needed for a document were included with that document. In SQL, you often needed to *join* across tables to build a full document.

However, recent revisions of the SQL standard have allowed JSON data to be stored in relational databases also. Some relational databases now let you store complex (non-scalar) data in table cells, and even search and index within them. JSON functions are supported in various ways for SQLite, PostgreSQL, MySQL, Oracle, and others.

This can be the best of both worlds. SQL databases have been around much longer, and have really useful features such as foreign keys and secondary indexes. Also, SQL is fairly standardized up to a point, and NoSQL query languages are all different.

Finally, there are new data design and query languages that try to combine SQL and NoSQL advantages, like EdgeQL that I mentioned earlier.

So, if you can’t fit your data into the rectangular relational box, look at a NoSQL database, a relational database with JSON support, or a “Post-SQL” database.

Database Load Testing

This book is mainly about FastAPI, but websites are so frequently tied tp databases

The data examples in this book have been tiny. To really stress-test a database, millions of items would be great. Rather than think of things to add, it’s easier use a package like faker. Faker can generate many kinds of data quickly — names, places, or special types that you define.

To generate lots of data, the Python package faker is quite handy. It can generate data of many types. In Example 17-3, `faker` pumps out names and countries, which are then loaded by `load()` into SQLite.

Example 14-3. Load fake explorers in test_load.py

```
from faker import Faker
from time import perf_counter

def load():
    from error import Duplicate
    from data.explorer import create
    from model.explorer import Explorer

    f = Faker()
    NUM = 100_000
    t1 = perf_counter()
    for row in range(NUM):
        try:
            create(Explorer(name=f.name(), nationality=f.country()))
        except Duplicate:
            pass
    t2 = perf_counter()
    print(NUM, "rows")
    print("write time:", t2-t1)

def read_db():
    from data.explorer import get_all

    t1 = perf_counter()
    data = get_all()
    t2 = perf_counter()
    print("db read time:", t2-t1)

def read_api():
    from fastapi.testclient import TestClient
    from main import app

    t1 = perf_counter()
    client = TestClient(app)
    resp = client.get("/explorer/")
    t2 = perf_counter()
    print("api read time:", t2-t1)

load()
read_db()
read_db()
read_api()
```

I’m catching the `Duplicate` exception in `load()` and ignoring it, because faker generates names from a limited list and is likely to repeat one now and then. So the result may be less than a million explorers loaded.

Also, I’m calling `read_db()` twice, to remove any startup time as SQLite does the query. Then `read_api()` timing should be fair. Example 17-4 fires it up.

Example 14-4. Test database query performance

```
$ python test_load.py
100000 rows
write time: 14.868232927983627
db read time: 0.4025074450764805
db read time: 0.39750714192632586
api read time: 2.597553930943832
```

The API read time for all explorers was much slower than the Data layer’s read time. Some of this is probably overhead from FastAPI’s conversion of the response to JSON. Also, the initial write time to the database wasn’t very zippy. It wrote one explorer at a time, because the Data layer API has a single `create()` function, but not a `create_many()`; on the read side, the API can return one (`get_one()`) or all (`get_all()`). So, if you ever want to do bulk loading, it might be good to add a new Data load function and a new Web endpoint (with restricted authorization).

Also, if you expect any table in your database to grow to 100,000 rows, maybe you shouldn’t allow random users to get all of them in one API call. Pagination would be useful, or a way to download a single CSV file from the table.

Data Science and AI

Python has become the most prominent language in data science in general, and machine learning in particular. So much data massaging is needed, and Python is good at that.

Sometimes, developers have used external tools like Pandas to do the data manipulation that’s too tricky in SQL.

Pytorch is one of the most popular ML tools, because it leverages Python’s strengths in data manipulation. The underlying computations may be in C or C++ for speed, but Python or Golang are well-suited for the “higher” data integration tasks. (The Mojo language, a superset of Python, may handle both the high and low ends if it succeeds as planned. Although a general purpose language, it specifically addresses some the current complexity in AI development.)

A new Python tool called ChromaDB is a database, similar to SQLite, but tailored to machine learning, specifically *LLMs* (large language models). Read the getting started page to, you know, get started.

Although AI development is complex and moving very fast, you can try out some AI with Python on your own machine without spending the megabucks that were behind GPT-4 and ChatGPT. Let's build a small FastAPI web interface to a small AI model.

NOTE

Model has different meanings in AI and Pydantic/FastAPI.

Hugging Face provides free AI models, datasets, and Python code to use them. First, install PyTorch and Hugging Face code:

```
$ pip install torch torchvision
$ pip install transformers
```

Example 14-5 shows a FastAPI application that uses Hugging Face's `transformers` module to access a pretrained mid-sized open source machine language model, and try to answer your prompts. (This was adapted from a command line example on the Youtube channel CodeToTheMoon.)

Example 14-5. Top-level LLM test (ai.py)

```python
from fastapi import FastAPI

app = FastAPI()

from transformers import (AutoTokenizer,
    AutoModelForSeq2SeqLM, GenerationConfig)
model_name = "google/flan-t5-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
config = GenerationConfig(max_new_tokens=200)
```

```
@app.get("/ai")
def prompt(line: str) -> str:
    tokens = tokenizer(line, return_tensors="pt")
    outputs = model.generate(**tokens,
        generator_config=config)
    result = tokenizer.batch_decode(outputs,
        skip_special_tokens=True)
    return result[0]
```

Run this with `uvicorn ai:app` (as always, first make sure you don't have another web server still running on localhost, port 8000). Feed it questions and get answers, like this (note the double == for an Httpie query parameter):

```
$ http -b localhost:8000/ai line=="What are you?"
"a sailor"
```

This is a fairly small model, and as you can see, it doesn't answer questions especially well. I tried other prompts (`line` arguments) and got equally noteworthy answers:

- Q: Are cats better than dogs?
- A: no
- Q: What does bigfoot eat for breakfast?
- A: a squid
- Q: Who comes down the chimney?
- A: a squealing pig
- Q: What group was John Cleese in?
- A: the Beatles
- Q: What has nasty pointy teeth?
- A: a teddy bear

These questions may get different answers at different times! Once it said that Bigfoot eats sand for breakfast. In AI-speak, answers like this are called *hallucinations*. You can get better answers by using a larger model, like `google/flan-75-xl`, but it will take longer to download model data and respond on a personal computer. And of course, models like ChatGPT that were trained on all the data they could find (using every CPU, GPU, TPU, and any other kind of PU), and will give excellent answers.

Review

This chapter expanded on the use of SQLite in Chapter 10, to other SQL databases, and even NoSQL ones. It also showed how some SQL databases can do NoSQL tricks with JSON support. Finally, it talked about the uses of database and special data tools have become more important as machine learning continues its explosive growth.

Chapter 15. Files

Preview

Besides fielding API requests and traditional content like HTML, web servers are expected to handle file transfers in both directions. Very large files may need to be transfered in *chunks* that don’t use too much of the system’s memory.

You can also provide access to a directory of files (ands subdirectories, to any depth) with `StaticFiles`.

Multipart Support

To handle large files, FastAPI's uploading and downloading features need these extra modules:

- python-multipart: `pip install python-multipart`
- aio-files: `pip install aiofiles`

Uploading Files

FastAPI targets API development, and most of the examples in this book have used JSON requests and responses. But in the last chapter you saw forms, which are handled differently. This chapter covers files, which are treated like forms in some ways.

FastAPI offers two techniques for file uploads: `File()` and `UploadFile`.

File()

`File()` is used as the type for a direct file upload. Your path function may be synchronous (`def`) or asynchronous (`async def`), but the asynchronous version is better because it won't tie up your web server while the file is uploading.

FastAPI will pull the file up in chunks and reassemble it in memory, so `File()` should only be used for relatively small files. Instead of assuming that the input is JSON, FastAPI encodes a file as a form element.

Let's write the code to request a file, and test it. You can grab any file on your machine to test with, or download one from a site like fastest.fish. I grabbed a 1K file from there, and saved it locally as *1KB.bin*.

In Example 19-1, add these lines to your top *main.py*:

Example 15-1. Handle a small file upload with FastAPI

```
@app.post("/small")
async def upload_small_file(small_file: bytes = File()) -> str:
    return f"file size: {len(small_file)}"
```

After Uvicorn restarts, try an Httpi test in Example 19-2:

Example 15-2. Upload a small file with Httpie.

```
$ http -f -b POST http://localhost:8000/small small_file@1KB.bin
"file size: 1000"
```

Some notes:

- Needs a `-f` (or `--form`), because files are uploaded like forms, not as JSON text.
- `small_file@1KB.bin`:
  - `small_file`: Matches the variable name `small_file` in the FastAPI path function in Example 19-1.
  - `@`: Httpie's shorthand to make a form
  - `1KB.bin`: The file that is being uploaded.

Example 19-3 is an equivalent programmatic test:

Example 15-3. Upload a small file with Requests.

```
$ python
>>> import requests
>>> url = "http://localhost:8000/small"
>>> files = {'small_file': open('1KB.bin', 'rb')}
>>> resp = requests.post(url, files=files)
>>> print(resp.json())
file size: 1000
```

UploadFile

For large files, it's better to use an `UploadFile`. This creates a Python `SpooledTemporaryFile` object, mostly on the server's disk instead of in memory. This is a Python *file-like* object, which supports the methods `read()`, `write()`, and `seek()`. Example 19-4 shows this, and also uses `async def` instead of `def` to avoid blocking the web server while file pieces are uploading:

Example 15-4. Upload a big file with FastAPI

```
from fastapi import UploadFile
```

```
@app.post("/big")
async def upload_big_file(big_file: UploadFile) -> str
    return f"file size: {big_file.size}, name: {big_file.filename}"
```

NOTE

`File()` created a `bytes` object and needed the parentheses. `UploadFile` is a different class of object.

If Uvicorn's starter motor isn't worn out yet, it's test time. This time, Examples 19-5 and 19-6 use a one gigabyte file (*1GB.bin*) that I grabbed from *fastest.fish*.

Example 15-5. Test big file upload with Httpie.

```
$ http -f -b POST http://localhost:8000/big big_file@1GB.bin
"file size: 1000000000, name: 1GB.bin"
```

Example 15-6. Test big file upload with Requests.

```
>>> import requests
>>> url = "http://localhost:8000/big"
>>> files = {'big_file': open('1GB.bin', 'rb')}
>>> resp = requests.post(url, files=files)
>>> print(resp.json())
file size: 1000000000, name: 1GB.bin
```

Downloading Files

Sadly, gravity doesn't make files download faster. Instead, there are equivalents of the upload methods.

FileResponse

First, in example 19-7, is the all-at-once version, `FileResponse`:

Example 15-7. Download a small file with `FileResponse`

```
from fastapi import FastAPI
from fastapi.responses import FileResponse

app = FastAPI()

@app.get("/small/{name}")
```

```
async def download_small_file():
    return FileResponse(name)
```

There’s a test around here somewhere. First, put the file *1KB.bin* in the same directory as *main.py*. Now, Example 19-8:

Example 15-8. Download a small file with Httpie

```
$ http -b http://localhost:8000/small/1KB.bin

-----------------------------------------
| NOTE: binary data not shown in terminal |
-----------------------------------------
```

If you don’t trust that suppression message, Example 19-9 pipes the output to a utility like `wc` to ensure that you got 1,000 bytes back:

Example 15-9. Download a small file with Httpie, with byte count

```
$ http -b http://localhost:8000/small/1KB.bin | wc -c
    1000
```

StreamingResponse

Similar to `FileUpload`, it’s better to download large files with `StreamingResponse`, which returns the file in chunks. Example 19-10 shows this, with an `async def` path function to avoid blocking when the CPU isn’t being used. I’m skipping error checking for now; if the file `path` doesn’t exist, the `open()` call will raise an exception.

Example 15-10. Return a big file with `StreamingResponse`

```
from typing import Generator
from fastapi.responses import StreamingResponse

def gen_file(path: str) -> Generator:
    with open(file=path, mode="rb") as file:
        yield file.read()

@app.get("/download_big/{name}")
async def download_big_file(name:str){
    gen_expr = gen_file(file_path=path)
    response = StreamingResponse(
        content=gen_expr,
        status_code=200,
    )
    return response
```

`gen_expr` is the *generator expression* returned by the *generator function* `gen_file()`. `StreamingResponse` uses it for its iterable `content` argument, so it can download the file in chunks.

Example 19-11 is the accompanying test. (This first needs the file *1GB.bin* alongside *main.py*, and will take a *little* longer.)

Example 15-11. Download a big file with Httpie

```
$ http -b http://localhost:8000/big/1GB.bin | wc -c
 1000000000
```

Serving Static Files

Traditional web servers can treat server files as though they were on a normal filesystem. FastAPI lets you do this with `StaticFiles`.

For this example, let's make a directory of (boring) free files for users to download.

- Make a directory called *static*, at the same level as *main.py*. (This can have any name, I'm only calling it *static* to help remember why I made it.)
- Put a text file called *abc.txt* in it, with the text contents `abc :)`.

Example 19-12 will serve any URL that starts with */static* (I could also have used any text string here) with files from the *static* directory:

Example 15-12. Serve everything in a directory with `StaticFiles`

```python
from pathlib import Path
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles

app = FastAPI()

# Directory containing main.py:
top = Path(__file__).resolve.parent

app.mount("/static",
    StaticFiles(directory=f"{top}/static", html=True),
    name="free")
```

That `top` calculation ensures I put `static` alongside *main.py*. The `__file__` variable is the full pathname of this file (*main.py*).

Example 19-13 is one way to manually test it:

Example 15-13. Get a static file

```
$ http -b localhost:8000/static/abc.txt
abc :)
```

What about that `html=True` argument that I passed to `StaticFiles()`? That makes it work a little more like a traditional server, returning an *index.html* file if one exists in that directory but you didn't ask for *index.html* explicitly in the URL. So, let's create an *index.html* file in the *static* directory with the contents `Oh. Hi!`, then test with Example 19-11:

Example 15-14. Get an index.html file from /static

```
$ http -b localhost:8000/static/
Oh. Hi!
```

You can have as many files (and subdirectories with files, etc.) as you want. Make a subdirectory *xyz* under *static*, and put two files there:

- *xyx.txt*: Contains the text `xyz :(`.
- *index.html*: Contains the text `How did you find me?`

I won't include the examples here. Try them yourself, with I hope more naming imagination.

Review

This chapter showed how to upload or download files — small, large, even gigantiferous. Plus, how to serve *static files* in nostalgic (non-API) web style from a directory.

Chapter 16. Forms and Templates

Preview

Although the *API* in FastAPI is a hint of its main focus, FastAPI can also handle traditional web content. This chapter talks about standard HTML forms and templates for inserting data into HTML.

Forms

As you've seen, FastAPI was mainly designed to build APIs, and its default input is JSON. But that doesn't mean that it can't serve standard banana HTML, forms, and friends.

FastAPI supports data from HTML forms much as it does from other sources like `Query` and `Path`, using the `Form` dependency.

You'll need the package `python-multipart` for any FastAPI forms work, so `pip install python-multipart` if you need to. Also, the *static* directory from Chapter 18 will be needed to house the test forms in this chapter.

In Example 19-1, let's redo example 3-11, but provide the `who` value via a form instead of a JSON string. (Call this path function `greet2()` to avoid clobbering the old `greet()` path function if it's still around.) Add this to *main.py*:

Example 16-1. Get a value from a GET form

```
from fastapi import Form

@app.get("/who2")
def greet2(name: str = Form()):
    return f"Hello, {name}?"
```

The main difference is that the value comes from `Form` instead of `Path`, `Query`, and the others from Chapter 3.

Try an initial form test with Httpie in Example 19-2 (you need that `-f` to upload with form encoding rather than as JSON):

Example 16-2. Form GET request with Httpie

```
$ http -f -b GET localhost:8000/who2 name="Bob Frapples"
"Hello, Bob Frapples?"
```

You could also send a request from a normal HTML form file. Chapter 18 showed how to make a directory called *static* (accessed under the URL */static*) that could house anything, including HTML files, so in Example 19-3 let's put this file (*form1.html*) there:

Example 16-3. Form GET request (static/form1.html)

```
<form action="http://localhost:8000/who2" method="get">
Say hello to my little friend:
<input type="text" name="name" value="Bob Frapples">
<input type="submit">
</form>
```

If you ask your browser to load *http://localhost:8000/static/form1.html*, you'll see a form. If you fill in any test string, you'll get this back:

```
  "detail":[{"loc":["body","name"],
             "msg":"field required",
             "type":"value_error.missing"}]}
```

Huh?

Look at the window where Uvicorn is running to see what its log says:

```
INFO:     127.0.0.1:63502 -
  "GET /who2?name=rr23r23 HTTP/1.1"
  422 Unprocessable Entity
```

Why did it send `name` as a query parameter when we had it in a form field? That turns out to be an HTML weirdness, documented here. Also, if you had any query parameters in your URL, it will erase them and replace them with `name`.

So, why did Httpie handle it as expected? I don't know. It's an inconsistency to be aware of.

The official HTML incantation is to change the action from a `GET` to a `POST`. So let's add a `POST` endpoint for */who2* to *main.py* in Example 19-4.

Example 16-4. Get a value from a POST form

```
from fastapi import Form

@app.post("/who2")
def greet3(name: str = Form()):
    return f"Hello, {name}?"
```

Example 19-5 is *stuff/form2.html*, with `get` changed to `post`.

Example 16-5. Form POST request (static/form2.html)

```
<form action="http://localhost:8000/who2" method="post">
Say hello to my little friend:
<input type="text" name="name">
<input type="submit">
</form>
```

Ask your browser to get off its digital haunches and get this new form for you. Fill in `Bob Frapples` and submit the form. This time, you'll get the result that you got from Httpie:

```
"Hello, Bob Frapples?"
```

So, if you're submitting forms from HTML files, use `POST`.

Templates

You may have seen *Mad Libs* — a game from a series of books. You ask people to provide a sequence of words — nouns, verbs, or something more specific —

and you enter them into labeled places in a page of text. Once you have all the words, you read the text with the inserted values, and hilarity ensures, sometimes with embarrassment.

Well, a web *template* is similar, though usually without the embarrassment. A template contains a bunch of text with slots for data to be inserted by the server. Its usual purpose is to generate HTML with variable content, unlike the *static* HTML of Chapter 18.

Users of Flask are very familiar with its companion project, the template engine Jinja (also often called Jinja2). FastAPI supports Jinja, as well as other template engines.

Make a directory called *template* alongside *main.py* to house Jinja-enhanced HTML files. Inside, make a file called *list.html*, as in Example 19-6:

Example 16-6. Define a template file (template/list.html)

```
<html>
<table bgcolor="#eeeeee">
  <tr>
    <th colspan=3>Creatures</th>
  </tr>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Location</th>
  </tr>
{% for creature in creatures: %}
  <tr>
    <td>{{ creature.name }}</td>
    <td>{{ creature.description }}</td>
    <td>{{ creature.location }}</td>
  </tr>
{% endfor %}
</table>

<br>

<table bgcolor="#dddddd">
  <tr>
    <th colspan=2>Explorers</th>
  </tr>
  <tr>
    <th>Name</th>
    <th>Nationality</th>
  </tr>
```

```
{% for explorer in explorers: %}
  <tr>
    <td>{{ explorer.name }}</td>
    <td>{{ explorer.nationality }}</td>
  </tr>
{% endfor %}
</table>
</html>
```

(I don't care how it looks, so there's no CSS, just the ancient pre-CSS `bgcolor` table attribute to distinguish the two tables.)

Double curly braces enclose Python variables that should be inserted, and `{%` and `%}` enclose `if` statements, `for` loops, and other controls. See the Jinja documentation for the syntax and examples.

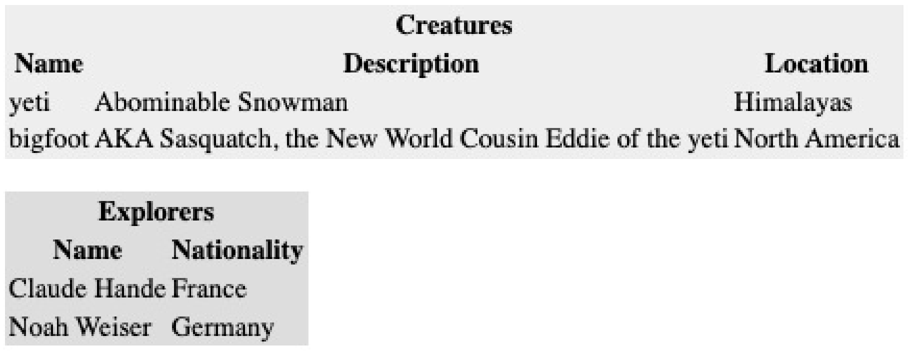
This template expects to be passed Python variables called `creatures` and `explorers`—lists of `Creature` and `Explorer` objects.

Example 19-7 shows what to add to *main.py* to set up templates and use the one from Example 19-6. It feeds `creatures` and `explorers` to it, using modules under the *fake* directory from previous chapters, which provided test data if the database was empty or not connected.

Example 16-7. Configure templates and use one (main.py)

```
from pathlib import Path
from fastapi.templating import Jinja2Templates
from fastapi import Request

app = FastAPI()

top = Path(__file__).resolve().parent

template_obj = Jinja2Templates(directory=f"{top}/template")

# Get some small predefined lists of our buddies:
from fake.creature import fakes as fake_creatures
from fake.explorer import fakes as fake_explorers

@app.get("/list")
def explorer_list(request: Request):
    return template_obj.TemplateResponse("list.html",
        {"request": request,
        "explorers": fake_explorers,
        "creatures": fake_creatures})
```

Ask your favorite browser, or even one that you don’t like very well, for *http://localhost:8000/list*, and you should get Example 19-8 back:



Creatures

Name	Description	Location
yeti	Abominable Snowman	Himalayas
bigfoot	AKA Sasquatch, the New World Cousin Eddie of the yeti	North America

Explorers

Name	Nationality
Claude Hande	France
Noah Weiser	Germany

Figure 16-1. Output from /list

Review

This chapter was a quick overview of how FastAPI handles non-API areas like forms and templates.

Appendix A. Further Reading

There are many great resources if you’d like to learn more, and fill in the areas that I didn’t cover in enough depth, or at all.

Python

Websites:

- Python.org — The mothership
- Real Python
- Reddit — Python subreddit
- Stackoverflow — Questions tagged “Python”
- Pycoder’s Weekly
- Anaconda — Scientific distribution

Books:

- *Introducing Python* (Bill Lubanovic, O’Reilly)

- *Python Distilled* (David Beazley, Addison-Wesley)
- *Fluent Python* (Luciano Ramalho, O’Reilly)
- *Robust Python* (Patrick Viafore, O’Reilly)
- *Architecture Patterns with Python* (Harry J. W. Percival and Bob Gregory, O’Reilly)

FastAPI

Websites:

- Home — The official site, and the best technical documentation that I’ve seen.
- External links and articles — From the official site.
- Github
- Awesome FastAPI
- The Ultimate FastAPI Tutorial
- The Blue Book: FastAPI
- Medium articles tagged “FastAPI”
- Using FastAPI to Build Python Web APIs
- Twitter
- Gitter
- Github

Even though FastAPI arrived late in 2018, not many books have popped up yet. Some that I’ve read and learned from are:

- *Bulding Data Science Applications with FastAPI* (François Voron, Packt)
- *Building Python Microservices with FastAPI* (Sherwin John C. Tegura, Packt)

- *Microservice APIs* (Josè Haro Peralta, Manning)

Starlette

- Home
- Github

Pydantic

- Home
- Docs
- Github

About the Author

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