



Building Data Science Applications with FastAPI

Develop, manage, and deploy efficient machine learning applications with Python



Building Data Science Applications with FastAPI

Develop, manage, and deploy efficient machine learning applications with Python

François Voron



BIRMINGHAM—MUMBAI

Building Data Science Applications with FastAPI

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Publishing Product Manager: Ashish Tiwari

Senior Editor: Mohammed Yusuf Imaratwale

Content Development Editor: Nazia Shaikh

Technical Editor: Manikandan Kurup

Copy Editor: Safis Editing

Project Coordinator: Aparna Nair

Proofreader: Safis Editing

Indexer: Subalakshmi Govindhan

Production Designer: Ponraj Dhandapani

First published: October 2021

Production reference: 1250821

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80107-921-1

www.packt.com

Contributors

About the author

François Voron is graduated from the University of Saint-Étienne (France) and the University of Alicante (Spain) with a master's degree in Machine Learning and Data Mining. A full-stack web developer and a data scientist, François has a proven track record working in the SaaS industry, with a special focus on Python backends and REST API.

He is also the creator and maintainer of FastAPI Users, the #1 authentication library for FastAPI, and is one of the top experts in the FastAPI community.

About the reviewers

Prajwal Nijhara is an electrical engineering student at Aligarh Muslim University and a member of the AUV-ZHCET club where he works on computer vision. He is a mentor at AMU-OSS and has worked with DeepSource as the developer-relation intern for six months.

*I dedicate my contribution in this book to my mentor **Areeb Jamal** who recently passed away recently. Areeb is forever in our hearts and our memory.*

*I also dedicate this book to **Vivek Duneja**, who taught me Python programming during my school days; in case, if you find any issues with my review, Mr. Duneja will be responsible.*

Finally, I'd like to thank my brother, parents, and family, for being supportive.

Izabela dos Santos Guerreiro graduated in information technology management and systems analysis and development. A machine learning enthusiast, Izabela is currently a postgraduate in artificial intelligence, machine learning, and data science. A software developer for 7 years, always working with Python, Izabela worked with FastAPI for about 1 year and became an enthusiast of the framework, collaborating on the translation of the documentation to her native language, Portuguese. She has already organized PyLadies and Django Girls events and is currently in charge of the Django Girls São Paulo organization.

Richad Becker leads data science at Revantage, a real estate shared service organization in the Blackstone family. Richad grew his career in real estate and finance data science – first at CoreLogic, focusing on text mining applications, then at Greystone, leading all things data in an innovation lab, before joining Revantage. Richad is self-taught in data science and credits his career progress to focusing on impact and deeply understanding the business case. Richad has a BA in neuroscience and anthropology and a master's in commerce (with a concentration in marketing and management) from the University of Virginia.

I'd like to thank my wife for her patience and support, and Jon for giving me advice on good technical reviewing!

William Jamir Silva is a software developer with a BSc in electrical engineering with more than 5 years of experience working in scientific software development with Python. He is skilled in desktop development and web applications.

Table of Contents

Preface

Section 1: Introduction to Python and FastAPI

1

Python Development Environment Setup

Technical requirements	4	Installing Python packages with pip	8
Installing a Python distribution using pyenv	4	Installing the HTTPie command-line utility	9
Creating a Python virtual environment	7	Summary	11

2

Python Programming Specificities

Technical requirements	14	Operating over sequences – list comprehensions and generators	34
Basics of Python programming	14	List comprehensions	34
Running Python scripts	14	Generators	36
Indentation matters	16	Writing object-oriented programs	39
Working with built-in types	17	Defining a class	39
Working with data structures – lists, tuples, dictionaries, and sets	18	Implementing magic methods	41
Performing Boolean logic and checking for existence	23	Reusing logic and avoiding repetition with inheritance	45
Controlling the flow of a program	25		
Defining functions	29		
Writing and using packages and modules	31		

Type hinting and type checking with mypy	48	Type function signatures with Callable	53
Getting started	48	Any and cast	54
The typing module	50	Asynchronous I/O	56
		Summary	60

3

Developing a RESTful API with FastAPI

Technical requirements	62	The request object	87
Creating the first endpoint and running it locally	62	Customizing the response	88
Handling request parameters	65	Path operation parameters	88
Path parameters	65	The response parameter	95
Query parameters	72	Raising HTTP errors	99
The request body	74	Building a custom response	102
Form data and file uploads	79	Structuring a bigger project with multiple routers	107
Headers and cookies	85	Summary	111

4

Managing Pydantic Data Models in FastAPI

Technical requirements	114	Applying validation at a field level	129
Defining models and their field types with Pydantic	114	Applying validation at an object level	130
Standard field types	114	Applying validation before Pydantic parsing	131
Optional fields and default values	120	Working with Pydantic objects	132
Field validation	122	Converting an object into a dictionary	132
Validating email addresses and URLs with Pydantic types	124	Creating an instance from a sub-class object	135
Creating model variations with class inheritance	126	Updating an instance with a partial one	137
Adding custom data validation with Pydantic	129	Summary	139

5

Dependency Injections in FastAPI

Technical requirements	142	Using dependencies at a path, router, and global level	152
What is dependency injection?	142	Use a dependency on a path decorator	153
Creating and using a function dependency	143	Use a dependency on a whole router	154
Get an object or raise a 404 error	147	Use a dependency on a whole application	156
Creating and using a parameterized dependency with a class	148	Summary	157
Use class methods as dependencies	150		

Section 2: Build and Deploy a Complete Web Backend with FastAPI

6

Databases and Asynchronous ORMs

Technical requirements	162	Setting up a database migration system with Alembic	180
An overview of relational and NoSQL databases	162	Communicating with a SQL database with Tortoise ORM	186
Relational databases	163	Creating database models	186
NoSQL databases	164	Setting up the Tortoise engine	188
Which one should you choose?	165	Creating objects	190
Communicating with a SQL database with SQLAlchemy	166	Updating and deleting objects	193
Creating the table schema	168	Adding relationships	194
Connecting to a database	169	Setting up a database migration system with Aerich	198
Making insert queries	171	Communicating with a MongoDB database using Motor	200
Making select queries	173	Creating models compatible with MongoDB ID	201
Making update and delete queries	175		
Adding relationships	177		

Connecting to a database	202	Updating and deleting documents	207
Inserting documents	203	Nesting documents	208
Getting documents	204	Summary	210

7

Managing Authentication and Security in FastAPI

Technical requirements	212	Implementing a database access token	220
Security dependencies in FastAPI	212	Implementing a login endpoint	222
Storing a user and their password securely in a database	216	Securing endpoints with access tokens	225
Creating models and tables	217	Configuring CORS and protecting against CSRF attacks	227
Hashing passwords	218	Understanding CORS and configuring it in FastAPI	228
Implementing registration routes	219	Implementing double-submit cookies to prevent CSRF attacks	233
Retrieving a user and generating an access token	220	Summary	239

8

Defining WebSockets for Two-Way Interactive Communication in FastAPI

Technical requirements	242	Handling concurrency	247
Understanding the principles of two-way communication with WebSockets	242	Using dependencies	250
Creating a WebSocket with FastAPI	243	Handling multiple WebSocket connections and broadcasting messages	253
		Summary	260

9

Testing an API Asynchronously with pytest and HTTPX

Technical requirements	262	Generating tests with parametrize	265
Introduction to unit testing with pytest	263	Reusing test logic by creating fixtures	267

Setting up testing tools for FastAPI with HTTPX	270	Testing with a database	278
Writing tests for REST API endpoints	275	Writing tests for WebSocket endpoints	286
Writing tests for POST endpoints	276	Summary	289

10

Deploying a FastAPI Project

Technical requirements	292	Adding database servers	303
Setting and using environment variables	292	Deploying a FastAPI application with Docker	304
Using a .env file	296	Writing a Dockerfile	304
Managing Python dependencies	297	Building a Docker image	306
Adding Gunicorn as a server process for deployment	299	Running a Docker image locally	307
Deploying a FastAPI application on a serverless platform	300	Deploying a Docker image	307
		Deploying a FastAPI application on a traditional server	309
		Summary	310

Section 3:

Build a Data Science API with Python and FastAPI

11

Introduction to NumPy and pandas

Technical requirements	314	Comparing arrays	325
Getting started with NumPy	314	Getting started with pandas	326
Creating arrays	315	Using pandas Series for one-dimensional data	326
Accessing elements and sub-arrays	318	Using pandas DataFrames for multi-dimensional data	328
Manipulating arrays with NumPy – computation, aggregations, comparisons	320	Importing and exporting CSV data	331
Adding and multiplying arrays	322	Summary	332
Aggregating arrays – sum, min, max, mean...	324		

12

Training Machine Learning Models with scikit-learn

Technical requirements	334	Classifying data with Naive Bayes models	346
What is machine learning?	334	Intuition	346
Supervised versus unsupervised learning	334	Classifying data with Gaussian Naive Bayes	347
Model validation	335	Classifying data with Multinomial Naive Bayes	350
Basics of scikit-learn	337	Classifying data with support vector machines	351
Training models and predicting	337	Intuition	352
Chaining pre-processors and estimators with pipelines	340	Using SVM in scikit-learn	355
Validating the model with cross-validation	344	Finding the best parameters	356
		Summary	358

13

Creating an Efficient Prediction API Endpoint with FastAPI

Technical requirements	360	Implementing an efficient prediction endpoint	363
Persisting a trained model with Joblib	360	Caching results with Joblib	366
Dumping a trained model	360	Choosing between standard or async functions	369
Loading a dumped model	362	Summary	372

14

Implement a Real-Time Face Detection System Using WebSockets with FastAPI and OpenCV

Technical requirements	374	Implementing a WebSocket to perform face detection on a stream of images	381
Getting started with OpenCV	374		
Implementing an HTTP endpoint to perform face detection on a single image	378		

Sending a stream of images from the browser in a WebSocket	383	Showing the face detection results in the browser Summary	387 390
--	-----	---	------------

Other Books You May Enjoy

Index

Preface

FastAPI is a web framework for building APIs with Python 3.6 and its later versions based on standard Python type hints. With this book, you'll be able to create fast and reliable data science API backends using practical examples.

This book starts with the basics of the FastAPI framework and associated modern Python programming concepts. You'll then be taken through all the aspects of the framework, including its powerful dependency injection system and how you can use it to communicate with databases, implement authentication, and integrate machine learning models. Later, you will cover best practices relating to testing and deployment to run a high-quality and robust application. You'll also be introduced to the extensive ecosystem of Python data science packages. As you progress, you'll learn how to build data science applications in Python using FastAPI. The book also demonstrates how to develop fast and efficient machine learning prediction backends and test them to achieve the best performance. Finally, you'll see how to implement a real-time face detection system using WebSockets and a web browser as a client.

By the end of this FastAPI book, you'll have not only learned how to implement Python in data science projects but also how to maintain and design them to meet high programming standards with the help of FastAPI.

Who this book is for

This book is for data scientists and software developers interested in gaining knowledge of FastAPI and its ecosystem to build data science applications. Basic knowledge of data science and machine learning concepts and how to apply them in Python is recommended.

What this book covers

Chapter 1, Python Development Environment Setup, is aimed at setting up the development environment so that you can start working with Python and FastAPI. We'll introduce the various tools that are commonly used in the Python community to ease development.

Chapter 2, Python Programming Specificities, introduces you to the specificities of programming in Python, specifically, block indentation, control flow statements, exceptions handling, and the object-oriented paradigm. We'll also cover features such as list comprehensions and generators. Finally, we'll see how type hinting and asynchronous I/O work.

Chapter 3, Developing a RESTful API with FastAPI, covers the basics of the creation of a RESTful API with FastAPI: routing, parameters, request body validation, and response. We'll also show how to properly structure a FastAPI project with dedicated modules and separate routers.

Chapter 4, Managing pydantic Data Models in FastAPI, covers in more detail the definition of data models with Pydantic, the underlying data validation library used by FastAPI. We'll explain how to implement variations of the same model without repeating ourselves thanks to class inheritance. Finally, we'll show how to implement custom data validation logic on those models.

Chapter 5, Dependency Injections in FastAPI, explains how dependency injection works and how we can define our own dependencies to reuse logic across different routers and endpoints.

Chapter 6, Databases and Asynchronous ORMs, demonstrates how we can set up a connection with a database to read and write data. We'll cover how to use two libraries to work asynchronously with SQL databases and how they interact with the Pydantic model. Finally, we'll also show you how to work with MongoDB, a NoSQL database.

Chapter 7, Managing Authentication and Security in FastAPI, shows us how to implement a basic authentication system to protect our API endpoints and return the relevant data for the authenticated user. We'll also talk about the best practices around CORS and how to be safe from CSRF attacks.

Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI, is aimed at understanding WebSockets and how to create them and handle the messages received with FastAPI.

Chapter 9, Testing an API Asynchronously with pytest and HTTPX, shows us how to write tests for our REST API endpoints.

Chapter 10, Deploying a FastAPI Project, covers the common configuration for running FastAPI applications smoothly in production. We'll also explore several deployment options: DigitalOcean App Platform, Docker, and the traditional server setup.

Chapter 11, Introduction to NumPy and pandas, introduces two core libraries for data science in Python: NumPy and pandas. We'll see how to create and manipulate arrays with NumPy and how we can do efficient operations on them. We'll then show how to manage large datasets with pandas.

Chapter 12, Training Machine Learning Models with scikit-learn, gives a quick introduction to machine learning before moving on to the scikit-learn library, a set of ready-to-use tools to perform machine learning tasks in Python. We'll review some of the most common algorithms and train prediction models.

Chapter 13, Creating an Efficient Prediction API Endpoint with FastAPI, shows us how we can efficiently store a trained machine learning model using Joblib. Then, we'll integrate it in a FastAPI backend, considering some technical details of FastAPI internals to achieve maximum performance. Finally, we'll show a way to cache results using Joblib.

Chapter 14, Implementing a Real-Time Face Detection System Using WebSockets with FastAPI and OpenCV, implements a simple application to perform face detection in the browser, backed by a FastAPI WebSocket and OpenCV, a popular library for computer vision.

To get the most out of this book

In this book, we'll mainly work with the Python programming language. The first chapter will explain how to set up a proper Python environment on your operating system. Some examples also involve running web pages with JavaScript, so you'll need a modern browser like Google Chrome or Mozilla Firefox.

Software/hardware covered in the book	Operating system requirements
Python 3.7 and above	Windows, macOS, or Linux
JavaScript	Windows, macOS, or Linux

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801079211_ColorImages.pdf

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Obviously, if everything is okay, we get a `Person` instance and have access to the properly parsed fields."

A block of code is set as follows:

```
from pydantic import BaseModel

class Person(BaseModel):
    first_name: str
    last_name: str
    age: int
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
class PostBase(BaseModel):
    title: str
    content: str

    def excerpt(self) -> str:
        return f"{self.content[:140]}..."
```

Any command-line input or output is written as follows:

```
1 validation error for Person
birthdate
  invalid date format (type=value_error.date)
```

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customer-care@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Building Data Science Applications with FastAPI*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: Introduction to Python and FastAPI

After setting up the development environment, we'll introduce the specificities of Python before starting to explore the basic features of FastAPI and running our first REST API.

This section comprises the following chapters:

- *Chapter 1, Python Development Environment Setup*
- *Chapter 2, Python Programming Specificities*
- *Chapter 3, Developing a RESTful API with FastAPI*
- *Chapter 4, Managing pydantic Data Models in FastAPI*
- *Chapter 5, Dependency Injections in FastAPI*

1

Python Development Environment Setup

Before we can get started on our FastAPI journey, we need to configure a clean and efficient Python environment. This chapter will show you the best practices and conventions that Python developers use daily to run their projects.

By the end of this chapter, you'll be able to run Python projects and install third-party dependencies in a contained environment that won't raise conflicts if you happen to work on another project that uses different versions of the Python language or different dependencies.

In this chapter, we're going to cover the following main topics:

- Installing a Python distribution using `pyenv`
- Creating a Python virtual environment
- Installing Python packages with `pip`
- Installing the HTTPie command-line utility

Technical requirements

Throughout this book, we'll assume you have access to a Unix-based environment, such as a Linux distribution or macOS.

If they haven't done so already, macOS users should install the Homebrew package (<https://brew.sh>), which helps a lot in installing command-line tools.

If you are a Windows user, you should enable **Windows Subsystem for Linux** (<https://docs.microsoft.com/windows/wsl/install-win10>), **WSL**, and install a Linux distribution (such as Ubuntu) that will run alongside the Windows environment, which should give you access to all the required tools. There are currently two versions of WSL, WSL and WSL2. Depending on your Windows version, you might not be able to install the newest version. However, we do recommend using WSL2 if your Windows installation supports it.

Installing a Python distribution using pyenv

Python is already bundled with most Unix environments. To ensure this is the case, you can run this command in a command line to show the version of the currently installed Python:

```
$ python3 --version
```

The output version displayed will vary depending on your system. You may think that this is enough to get started, but it poses an important issue: *you can't choose the Python version for your project*. Each Python version introduces new features and breaking changes. Thus, it's important to be able to switch to a recent version for new projects to take advantage of the new features but still be able to run older projects that may not be compatible. This is why we need `pyenv`.

`pyenv` (<https://github.com/pyenv/pyenv>) is a tool that helps you manage and switch between multiple Python versions on your system. It allows you to set a default Python version for your whole system but also per project.

Beforehand, you need to install several build dependencies on your system to allow `pyenv` to compile Python on your system. The official documentation provides clear guidance on this (<https://github.com/pyenv/pyenv/wiki#suggested-build-environment>), but here are the commands you should run:

1. Install the build dependencies:
 - macOS users, use this:

```
$ brew install openssl readline sqlite3 xz zlib
```

- Ubuntu users, use this:

```
$ sudo apt-get update; sudo apt-get install --no-install-recommends make build-essential libssl-dev zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm libncurses5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev
```

Package managers

Brew and APT are what are commonly known as package managers. Their role is to automate the installation and management of software on your system. Thus, you don't have to worry about where to download them and how to install and uninstall them. The commands just tell the package manager to update its internal package index and then install the list of required packages.

2. Install pyenv:

```
$ curl https://pyenv.run | bash
```

Tip

If you are a macOS user, you can also install it with Homebrew: `brew install pyenv`.

3. This will download and execute an installation script that will handle everything for you. At the end, it'll prompt you with some instructions to add some lines to your shell scripts so that pyenv is discovered properly by your shell:
 - a. Open your `~/.profile` script in nano, a simple command-line text editor:

```
$ nano ~/.profile
```

- b. Add the following lines before the block containing `~/.bashrc`:

```
export PYENV_ROOT="$HOME/.pyenv"
```

```
export PATH="$PYENV_ROOT/bin:$PATH"
```

```
eval "$(pyenv init --path)"
```

- c. Save by using the keyboard shortcut `Ctrl + O` and confirm by pressing `Enter`. Then, quit by using the keyboard shortcut `Ctrl + X`.

d. Open your `~/ .bashrc` script in nano. If you are using zsh instead of Bash (the default on the latest macOS), the file is named `~/ .zshrc`:

```
$ nano ~/.bashrc
```

e. Add the following line at the end:

```
eval "$(pyenv init -)"
```

f. Save by using the keyboard shortcut `Ctrl + O` and confirm by pressing `Enter`. Then, quit by using the keyboard shortcut `Ctrl + X`.

4. Reload your shell configuration to apply those changes:

```
$ source ~/.profile && exec $SHELL
```

5. If everything went well, you should now be able to invoke the `pyenv` tool:

```
$ pyenv
```

```
pyenv 1.2.21
```

```
Usage: pyenv <command> [<args>]
```

6. We can now install the Python distribution of our choice. Even though FastAPI is compatible with Python 3.6 and later, we'll use Python 3.7 throughout this book, which has more mature handling of the asynchronous paradigm. All the examples in the book were tested with this version but should work flawlessly with newer versions. Let's install Python 3.7:

```
$ pyenv install 3.7.10
```

This may take a few minutes since your system will have to compile Python from the source.

7. Finally, you can set the default Python version with the following command:

```
$ pyenv global 3.7.10
```

This will tell your system to always use Python 3.7.10 by default, unless specified otherwise in a specific project.

8. To make sure everything is in order, run the following command to check the Python version that is invoked by default:

```
$ python --version
```

```
Python 3.7.10
```

Congratulations! You can now handle any version of Python on your system and switch it whenever you like!

Creating a Python virtual environment

As for many programming languages of today, the power of Python comes from the vast ecosystem of third-party libraries, including **FastAPI**, of course, that help you build complex and high-quality software very quickly. The **Python Package Index** (<https://pypi.org>), **PyPi**, is the public repository that hosts all those packages. This is the default repository that will be used by the built-in Python package manager, `pip`.

By default, when you install a third-party package with `pip`, it will install it for the *whole system*. This is different from some other languages, such as Node.js' `npm`, which by default creates a local directory for the current project to install those dependencies. Obviously, this may cause issues when you work on several Python projects with dependencies that have conflicting versions. It also makes it difficult to retrieve only the dependencies necessary to deploy a project properly on a server.

This is why Python developers generally use **virtual environments**. Basically, a virtual environment is just a directory in your project containing a copy of your Python installation and the dependencies of your project. It's quite similar to the `node_modules` directory in Node.js. This pattern is so common that the tool to create them is bundled with Python:

1. Create a directory that will contain your project:

```
$ mkdir fastapi-data-science
```

```
$ cd fastapi-data-science
```

Tip

If you are on Windows with WSL, we recommend that you create your working folder on the Windows drive rather than the virtual filesystem of the Linux distribution. It'll allow you to edit your source code files in Windows with your favorite text editor or IDE while running them in Linux.

To do this, you can actually access your `C:` drive in the Linux command line through `/mnt/c`. You can thus access your personal documents using the usual Windows path, for example, `cd /mnt/c/Users/YourUsername/Documents`.

2. You can now create a virtual environment:

```
$ python -m venv
```

Basically, this command tells Python to run the `venv` package of the standard library to create a virtual environment in the `venv` directory. The name of this directory is a convention, but you can choose another name if you wish.

3. Once this is done, you have to activate this virtual environment. It'll tell your shell session to use the Python interpreter and the dependencies in the local directory instead of the global ones. Simply run the following command:

```
$ source venv/bin/activate
```

After doing this, you may notice that the prompt adds the name of the virtual environment:

```
(venv) $
```

Remember that the activation of this virtual environment is only available for the *current session*. If you close it or open other command prompts, you'll have to activate it again. This is quite easy to forget, but it will become natural after some practice with Python.

You are now ready to install Python packages safely in your project!

Installing Python packages with pip

As we said earlier, `pip` is the built-in Python package manager that will help us install third-party libraries. To get started, let's install **FastAPI** and **Uvicorn**:

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

We'll talk about it in later chapters, but Uvicorn is required to run a FastAPI project.

Tip

You have probably noticed the word `standard` inside square brackets just after `uvicorn`. Sometimes, some libraries have sub-dependencies that are not required to make the library work. Usually, they are needed for optional features or specific project requirements. The square brackets are here to indicate that we want to install the `standard` sub-dependencies of `uvicorn`.

To make sure the installation worked, we can open a Python interactive shell and try to import the `FastAPI` package:

```
$ python
>>> from fastapi import FastAPI
```

If it passes without any errors, congratulations, `FastAPI` is installed and ready to use!

Installing the HTTPie command-line utility

Before getting into the heart of the topic, there is one last tool that we'll install. `FastAPI` is, as you probably know, mainly about building **REST APIs**. To do so, you have several options:

- `FastAPI` automatic documentation (we'll talk about this later in the book)
- **Postman**, a GUI tool to perform HTTP requests
- **cURL**, the well-known and widely used command-line tool to perform network requests

Even if visual tools are nice and easy to use, they sometimes lack some flexibility and may not be as productive as command-line tools. On the other hand, `cURL` is a very powerful tool with thousands of options but can be complex and verbose for testing simple REST APIs.

This is why we'll introduce `HTTPie`, a command-line tool aimed at making HTTP requests with an intuitive syntax, JSON support, and syntax highlighting. It's available to install from most package managers:

- macOS users, use this:

```
$ brew install httpie
```

- Ubuntu users, use this:

```
$ sudo apt-get update; sudo apt-get install httpie
```

Let's see how to perform simple requests on a dummy API:

1. First, let's retrieve data:

```
$ http GET https://603cca51f4333a0017b68509.mockapi.io/
todos
HTTP/1.1 200 OK
Content-Length: 195
```

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

```
[  
  {  
    "id": "1",  
    "text": "Island"  
  }  
]
```

As you can see, you can invoke HTTPie with the `http` command and simply type the HTTP method and the URL. It outputs both the HTTP headers and the JSON body in a clean and formatted way.

2. HTTPie also supports sending JSON data in a request body very quickly without having to format the JSON yourself:

```
$ http -v POST https://603cca51f4333a0017b68509.mockapi.io/todos text="My new task"
```

```
POST /todos HTTP/1.1
```

```
Accept: application/json, */*;q=0.5
```

```
User-Agent: HTTPie/2.3.0
```

```
{  
  "text": "My new task"  
}
```

```
HTTP/1.1 201 Created
```

```
Content-Length: 31
```

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

```
{  
  "id": "6",  
  "text": "My new task"  
}
```

By simply typing the property name and its value separated by `=`, HTTPie will understand that it's part of the request body in JSON. Notice here that we specified the `-v` option, which tells HTTPie to *output the request* before the response, which is very useful to check that we properly specified the request.

3. Finally, let's see how we can specify *request headers*:

```
$ http -v GET https://603cca51f4333a0017b68509.mockapi.io/todos "My-Header: My-Header-Value"
GET /todos HTTP/1.1
Accept: */*
My-Header: My-Header-Value
User-Agent: HTTPie/2.3.0

HTTP/1.1 200 OK
Content-Length: 227
Content-Type: application/json

[
  {
    "id": "1",
    "text": "Island"
  }
]
```

That's it! Just type your header name and value separated by a colon to tell HTTPie it's a header.

Summary

You now have all the tools and setup required to confidently run the examples of this book and all your future Python projects. Understanding how to work with `pyenv` and virtual environments is a key skill to ensure everything goes smoothly when you switch to another project or when you have to work on somebody else's code. You also learned how to install third-party Python libraries using `pip`. Finally, you saw how to use HTTPie, a simple and efficient way to run HTTP queries that will make you more productive while testing your REST APIs.

In the next chapter, we'll highlight some of Python's peculiarities as a programming language and get a grasp of what it means *to be Pythonic*.

2

Python Programming Specificities

The Python language was designed to emphasize code readability. As such, it provides syntaxes and constructs that allow developers to quickly express complex concepts in few and readable lines. However, this makes it quite different from other programming languages.

The goal of this chapter is thus to get you acquainted with its specificities, but we expect you already have some experience with programming. We'll first get started with the basics of the language, the standard types, and the flow control syntaxes. You'll also be introduced to the list comprehension and generator concepts, which are very powerful ways to go through and transform sequences of data. You'll also see that Python can be used as an object-oriented language, still through a very lightweight yet powerful syntax. Before moving on, we'll also review the concepts of type hinting and asynchronous I/O, which are quite new in Python but are at the core of the **FastAPI** framework.

In this chapter, we're going to cover the following main topics:

- Basics of Python programming
- List comprehensions and generators
- Classes and objects
- Type hinting and type checking with `mypy`
- Asynchronous I/O

Technical requirements

You'll need a Python virtual environment, as we set up in *Chapter 1, Python Development Environment Setup*.

You'll find all the code examples of this chapter in the dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter2>.

Basics of Python programming

First of all, let's review some of the key aspects of Python:

- It's an **interpreted language**. Contrary to languages such as C or Java, it doesn't need to be compiled, which allows us to run Python code interactively.
- It's **dynamically typed**. The type of values is determined at runtime.
- It supports several **programming paradigms**: procedural, object-oriented, and functional programming.

This makes Python quite a versatile language, from simple automation scripts to complex data science projects.

Let's now write and run some Python!

Running Python scripts

As we said, Python is an interpreted language. Hence, the simplest and quickest way to run some Python code is to launch an interactive shell. Just run the following command to start a session:

```
$ python
```

```
Python 3.7.10 (default, Mar 7 2021, 10:12:14)
```

```
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

This shell makes it very easy to run some simple statements and make some experiments:

```
>>> 1 + 1
2
>>> x = 100
>>> x * 2
200
```

To exit the shell, use the *Ctrl + D* keyboard shortcut.

Obviously, this can become tedious when you start to have more statements or if you just wish to keep your work to reuse it later. Python scripts are saved in files with the `.py` extension. Let's create a file named `chapter2_basics_01.py` in our project directory and add this code:

chapter2_basics_01.py

```
print("Hello world!")
x = 100
print(f"Double of {x} is {x * 2}")
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2\_basics\_01.py
```

Quite simply, this script prints `Hello world` in the console, assigns the value `100` to a variable named `x`, and prints a string with the value of `x` and its double. To run it, simply add the path of your script as a parameter of the `python` command:

```
$ python chapter2_basics_01.py
Hello world!
Double of 100 is 200
```

f-strings

You have probably noticed the string starting with *f*. This syntax, called *f-strings*, is a very convenient and neat way to perform string interpolation. Within, you can simply insert variables between curly braces; they will automatically be converted into strings to build the resulting string. We'll use it quite often in our examples.

That's it! You are now able to write and run simple Python scripts. Let's now dive deeper into the Python syntax.

Indentation matters

One of the most iconic aspects of Python is that code blocks are not defined using curly braces like many other programming languages, but rather with **whitespace indentation**. This may sound a bit strange, but it's at the heart of the readability philosophy of Python. Let's see how you can write a script that finds the even numbers in a list:

chapter2_basics_02.py

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even = []
for number in numbers:
    if number % 2 == 0:
        even.append(number)

print(even) # [2, 4, 6, 8, 10]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_02.py

In this script, we define `numbers`, a list of numbers from 1 to 10, and `even`, an empty list that will contain the even numbers.

Then, we define a `for` loop statement to go through each element of `numbers`. As you see, we open a block with a colon, `:`, break a line, and start writing the next statement with an indentation.

The next line is a conditional statement to check the parity of the current number. Once again, we open a block with a colon, `:`, and write the next statement with an additional indentation level. This statement adds the even number to the `even` list.

After that, the next statements are not intended. This means that we are out of the `for` loop block; they should be executed after the iteration is finished.

Let's run it:

```
$ python chapter2_basics_02.py
[2, 4, 6, 8, 10]
```

Indentation style and size

You can choose the indentation style (tabs or spaces) and size (2, 4, 6...) you prefer; the only constraint is that you should be consistent *within* a block. However, by convention, Python developers usually go for a **four-space indentation**.

This aspect of Python may sound weird but with some practice, you'll find that it enforces clear formatting and greatly improves the readability of your scripts.

We'll now review the built-in types and data structures.

Working with built-in types

Python is quite conventional regarding scalar types. There are six of them:

- `int`, to store *integer* values, such as `x = 1`
- `float`, for *floating-point numbers*, such as `x = 1.5`
- `complex`, for *complex numbers*, such as `x = 1 + 2j`
- `bool`, for *Boolean* values, either `True` or `False`
- `str`, for *string* values, such as `x = "abc"`
- `NoneType`, to indicate *null* values, such as `x = None`

It's worth noting that Python is *strongly typed*, meaning that the interpreter will limit implicit type conversions. For example, trying to add an `int` value and a `str` value will raise an error, as you can see in the following example:

```
>>> 1 + "abc"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Still, adding an `int` value and a `float` value will automatically upcast the result to `float`:

```
>>> 1 + 1.5
2.5
```

As you may have noticed, Python is quite traditional regarding those standard types. Let's see now how basic data structures are handled.

Working with data structures – lists, tuples, dictionaries, and sets

Besides the scalar types, Python also provides handy data structures: an array structure, of course, called a *list* in Python, but also *tuples*, *dictionaries*, and *sets*, which are very convenient in lots of cases. Let's start with lists.

Lists

Lists are the equivalent in Python of the classic array structure. Defining a list is quite straightforward:

```
>>> l = [1, 2, 3, 4, 5]
```

As you can see, wrapping a suite of elements in *square brackets* denotes a list. You can of course access single elements by index:

```
>>> l[0]
1
>>> l[2]
3
```

It also supports *negative indexing*, which allows retrieving elements from the end of the list: index `-1` is the last element, `-2` is the second-last element, and so on:

```
>>> l[-1]
5
>>> l[-4]
2
```

Another useful syntax is slicing, which quickly allows you to retrieve a sub-list:

```
>>> l[1:3]
[2, 3]
```

The first number is the start index (inclusive) and the second one is the end index (exclusive), separated by a colon. You can omit the first one; in this case, 0 is assumed:

```
>>> l[:3]
[1, 2, 3]
```

You can also omit the second one; in this case, the length of the list is assumed:

```
>>> l[1:]
[2, 3, 4, 5]
```

Finally, this syntax also supports a third argument to specify the step size. It can be useful to select every second element of the list:

```
>>> l[::2]
[1, 3, 5]
```

A useful trick with this syntax is to use -1 to reverse the list:

```
>>> l[::-1]
[5, 4, 3, 2, 1]
```

Lists are **mutable**. This means that you can reassign elements or add new ones:

```
>>> l[1] = 10
>>> l
[1, 10, 3, 4, 5]
>>> l.append(6)
[1, 10, 3, 4, 5, 6]
```

This is different from their cousin, tuples, which are **immutable**.

Tuples

Tuples are very similar to lists. Instead of square brackets, they are defined using *parentheses*:

```
>>> t = (1, 2, 3, 4, 5)
```

They support the same syntax as lists to access elements or slicing:

```
>>> t[2]
3
>>> t[1:3]
(2, 3)
>>> t[::-1]
(5, 4, 3, 2, 1)
```

However, tuples are immutable. This means that you can't add new elements or change existing ones. Trying to do so will raise an error:

```
>>> t[1] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t.append(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

A common way to use them is for functions that have multiple return values. In the following example, we define a function to compute and return both the quotient and remainder of the Euclidean division:

chapter2_basics_03.py

```
def euclidean_division(dividend, divisor):
    quotient = dividend // divisor
    remainder = dividend % divisor
    return (quotient, remainder)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_03.py

This function simply returns the quotient and remainder wrapped in a tuple. Let's now compute the Euclidean division of 3 and 2:

chapter2_basics_03.py

```
t = euclidean_division(3, 2)
print(t[0]) # 1
print(t[1]) # 1
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_03.py

In this case, we assign the result to a tuple named `t`, and simply retrieve the quotient and remainder *by index*. However, we can do something better than that. Let's compute the Euclidean division of 42 and 4:

chapter2_basics_03.py

```
q, r = euclidean_division(42, 4)
print(q) # 10
print(r) # 2
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_03.py

You see here that we directly assign the quotient and remainder to the `q` and `r` variables, respectively. This syntax is called **unpacking** and is very convenient to assign variables from list or tuple elements. It's worth noting that since `t` is a tuple, it's immutable, so you can't reassign the values. On the other hand, `q` and `r` are new variables and therefore are mutable.

Dictionaries

A dictionary is also a widely used data structure in Python, to map keys to values. It is defined using curly braces, with a list of keys and values separated by a colon:

```
>>> d = {"a": 1, "b": 2, "c": 3}
```


Elements can be accessed by key:

```
>>> d["a"]  
1
```

Dictionaries are mutable, so you can reassign or add elements in the mapping:

```
>>> d["a"] = 10  
>>> d  
{'a': 10, 'b': 2, 'c': 3}  
>>> d["d"] = 4  
>>> d  
{'a': 10, 'b': 2, 'c': 3, 'd': 4}
```

Sets

A set is a convenient data structure to store a collection of unique items. It is defined using curly braces:

```
>>> s = {1, 2, 3, 4, 5}
```

Elements can be added to the set, but the structure ensures elements appear only once:

```
>>> s.add(1)  
>>> s  
{1, 2, 3, 4, 5}  
>>> s.add(6)  
{1, 2, 3, 4, 5, 6}
```

Convenient methods are also provided to perform operations such as unions or intersections on two sets:

```
>>> s.union({4, 5, 6})  
{1, 2, 3, 4, 5, 6}  
>>> s.intersection({4, 5, 6})  
{4, 5}
```

That's all for this overview of the Python data structures. You'll probably use them quite often in your programs, so take some time to get acquainted with them. Obviously, we didn't cover all of their methods and specificities, but you can have a look at the official Python documentation for exhaustive information: <https://docs.python.org/3/library/stdtypes.html>.

Let's now talk about the different types of operators available in Python that will allow us to perform some logic on this data.

Performing Boolean logic and checking for existence

Predictably, Python provides operators to perform Boolean logic. However, we'll also see that there are other operators that are less common but make Python a very efficient language to work with.

Performing Boolean logic

Boolean logic is performed with the `and`, `or`, and `not` keywords. Let's review some simple examples:

```
>>> x = 10
>>> x > 0 and x < 100
True
>>> x > 0 or (x % 2 == 0)
True
>>> not (x > 0)
False
```

You'll probably use them quite often in your programs, especially with conditional blocks. Let's now review the identity operators.

Checking whether two variables are the same

The `is` and `is not` identity operators check whether two variables *refer* to the same object. This is different from the `==` and `!=` comparison operators, which check whether two variables have the same *value*.

Internally, Python stores variables in pointers. The goal of the identity operators is thus to check whether two variables actually point to the same **pointer**. Let's review some examples:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Even though lists `a` and `b` are identical, they don't share the same pointer, so `a is b` is false. However, `a == b` is true. Let's see what happens if we assign `a` to `b`:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
```

In this case, the `b` variable will now refer to the same pointer as `a`, that is, the same list in memory. Thus, the identity operator is true.

is None or == None?

To check whether a variable is null, you could write `a == None`. While it will work most of the time, it's generally advised to write `a is None`.

Why? In Python, classes can implement custom comparison operators, so the result of `a == None` may be unpredictable in some cases, since a class can choose to attach a special meaning to the `None` value.

We'll now review the membership operators.

Checking whether a value is present in a data structure

The membership operators, `in` and `not in`, are very useful to check whether an element is present in data structures such as lists or dictionaries. They are idiomatic of Python and make this operation very efficient and easy to write. Let's review some examples:

```
>>> l = [1, 2, 3]
>>> 2 in l
True
>>> 5 not in l
True
```

With the membership operators, we can check in one statement whether an element is present or not in a list. It also works with tuples and sets:

```
>>> t = (1, 2, 3)
```

```
>>> 2 in t
```

```
True
```

```
>>> s = {1, 2, 3}
```

```
>>> 2 in s
```

```
True
```

Finally, it also works with dictionaries. In this case, the membership operators check whether the *key* is present, not the value:

```
>>> d = {"a": 1, "b": 2, "c": 3}
```

```
>>> "b" in d
```

```
True
```

```
>>> 3 in d
```

```
False
```

We are now clear about those common operations. We'll now put them to use with conditional statements.

Controlling the flow of a program

A programming language would not be a programming language without its control flow statements. Once again, you'll see that Python is a bit different from other languages. Let's start with conditional statements.

Executing operations conditionally – if, elif, else

Classically, those statements are there to perform some logic based on some Boolean conditions. In the following example, we'll consider a situation where we have a dictionary containing information about an e-commerce website order. We'll write a function that will change the order status to the next step given the current status:

chapter2_basics_04.py

```
def forward_order_status(order):
```

```
    if order["status"] == "NEW":
```

```
        order["status"] = "IN_PROGRESS"
```

```
    elif order["status"] == "IN_PROGRESS":
```

```
        order["status"] = "SHIPPED"
    else:
        order["status"] = "DONE"
    return order
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2\_basics\_04.py
```

The first condition is noted as `if`, followed by a Boolean condition. We then open an indented block, as we explained in the *Indentation matters* section of this chapter.

The alternate conditions are noted as `elif` (not `else if`) and the fallback block is noted as `else`. Of course, those are *optional* if you don't need alternate or fallback conditions.

It's also worth noting that, contrary to many other languages, Python does not provide a `switch` statement.

We'll now move on to another classic control flow statement: the `for` loop. You can repeat operations over a sequence using the `for` loop statement.

We already saw an example of the `for` loop in action in the *Indentation matters* section of this chapter. As you probably understood, this statement is useful for repeating the execution of a code block.

You also may have noticed that it works a bit differently than other languages. Usually, programming languages define `for` loops like this: `for (i = 0; i <= 10; i++)`. They give you the responsibility to define and control the variable used for the iteration.

Python doesn't work this way. Instead, it expects you to feed the loop with an **iterator**. An iterator can be seen as a sequence of elements that you can retrieve one by one. Lists, tuples, dictionaries, and sets can behave like an iterator and be used in a `for` loop. Let's see some examples:

```
>>> for i in [1,2,3]:
...     print(i)
...
1
2
3
>>> for k in {"a": 1, "b": 2, "c": 3}:
...     print(k)
```

```
...  
a  
b  
c
```

But what if you just wish to iterate a certain number of times? Luckily, Python has built-in functions that generate some useful iterators. The most known is `range`, which precisely creates a sequence of numbers. Let's see how it works:

```
>>> for i in range(3):  
...     print(i)  
...  
0  
1  
2
```

`range` will generate a sequence of the size you provided in the first argument, starting with 0.

You could also be more precise by specifying two arguments: the start index (inclusive) and the last index (exclusive):

```
>>> for i in range(1, 3):  
...     print(i)  
...  
1  
2
```

Finally, you may even provide a step as the third argument:

```
>>> for i in range(0, 5, 2):  
...     print(i)  
...  
0  
2  
4
```

Note that this syntax is quite similar to the slicing syntax we saw earlier in this chapter in the sections dedicated to *Lists* and *Tuples*.

range output is not a list

A common misconception is to think that `range` returns a *list*. It's actually a *sequence* object that only stores the `start`, `end`, and `step` arguments. That's why you could write `range(1000000000)` without blowing up your memory; the millions of integers are not assigned in memory all at once.

As you see, the `for` loop syntax in Python is quite straightforward to understand and emphasizes readability. We'll now have a word about its cousin, the `while` loop.

Repeating operations until a condition is met – the `while` loop statement

The classical `while` loop is also available in Python. At the risk of disappointing you, there is nothing truly special about this one. Classically, this statement allows you to repeat instructions until a condition is met. We'll review an example in which we use a `while` loop to retrieve paginated elements until we reach the end:

chapter2_basics_05.py

```
def retrieve_page(page):
    if page > 3:
        return {"next_page": None, "items": []}
    return {"next_page": page + 1, "items": ["A", "B", "C"]}

items = []
page = 1
while page is not None:
    page_result = retrieve_page(page)
    items += page_result["items"]
    page = page_result["next_page"]
print(items) # ["A", "B", "C", "A", "B", "C", "A", "B", "C"]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_05.py

The `retrieve_page` function is a dummy function that returns a dictionary with the items for the page passed in an argument and the next page number or `None` if we reached the last page. *A priori*, we don't know how many pages there are. Thus, we repeatedly call `retrieve_page` until page is `None`. At each iteration, we save the current page items in an accumulator, `items`.

This kind of use case is quite common when you are dealing with third-party REST APIs and you wish to retrieve all items available, and `while` loops perfectly help with this.

It may happen though that you wish for finer control of the loop behavior. This is where `break` and `continue` come in.

Finely controlling a loop – break and continue

There are cases where you wish to prematurely end the loop or skip an iteration. To solve this, Python implements the classic `break` and `continue` statements.

Defining functions

Now that we know how to use the common operators and control the flow of our program, let's put it in reusable logic. As you may have guessed, we'll look at **functions** and how to define them. We already saw them in some of our previous examples, but let's introduce them more formally.

In Python, functions are defined using the `def` keyword followed by the name of the function. Then, you have the list of supported arguments in parentheses, before a colon that indicates the start of the function body. Let's see a simple example:

```
>>> def f(a):  
...     return a  
...  
>>> f(2)  
2
```

That's it! Python also supports default values on arguments:

```
>>> def f(a, b = 1):  
...     return a, b  
...  
>>> f(2)  
(2, 1)  
>>> f(2, 3)  
(2, 3)
```

When calling a function, you can specify the values of arguments using their name:

```
>>> f(a=2, b=3)  
(2, 3)
```


Those arguments are called *keyword arguments*. They are especially useful if you have several default arguments but only wish to set one of them:

```
>>> def f(a = 1, b = 2, c = 3):
...     return a, b, c
...
>>> f(c=1)
(1, 2, 1)
```

Function naming

By convention, functions should be named using **snake case**: `my_wonderful_function` but not `MyWonderfulFunction`.

But there is more! You can actually define functions accepting a dynamic number of arguments.

Accepting arguments dynamically with `*args` and `**kwargs`

Sometimes, you may need a function that supports a dynamic number of arguments. Those arguments are then handled in your function logic at runtime. To do this, you have to use the `*args` and `**kwargs` syntax. Let's define a function that uses this syntax and prints the value of those arguments:

```
>>> def f(*args, **kwargs):
...     print("args", args)
...     print("kwargs", kwargs)
...
>>> f(1, 2, 3, a=4, b=5)
args (1, 2, 3)
kwargs {'a': 4, 'b': 5}
```

As you see, standard arguments are placed in a *tuple*, in the same order as they have been called. Keyword arguments, on the other hand, have been placed in a *dictionary*, with the key being the name of the argument. It's up to you then to use this data to perform your logic!

Interestingly, you can mix both approaches so that you have hardcoded arguments and dynamic ones:

```
>>> def f(a, *args):
...     print("a", a)
...     print("arg", args)
...
>>> f(1, 2, 3)
a 1
arg (2, 3)
```

Well done! You have learned how to write functions in Python to organize the logic of your program. The next step now is to organize those functions into modules and import them into other modules to take advantage of them!

Writing and using packages and modules

You probably already know that, apart from small scripts, your source code shouldn't live in one big file with thousands of lines. Instead, you should split it into logical blocks of reasonable size that are easy to maintain. That's exactly what packages and modules are for! We'll see how they work and how you can define your own.

First of all, Python comes with its own set of modules, the standard library, that are directly importable in a program:

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2021, 3, 12)
```

With just the `import` keyword, you can use the `datetime` module and access all its content by referring to its namespace, `datetime.date`, which is the built-in class to work with dates. However, you may wish sometimes to explicitly import a part of this module:

```
>>> from datetime import date
>>> date.today()
datetime.date(2021, 3, 12)
```

Here, we explicitly import the `date` class to use it directly.

The same principles apply to third-party packages installed with `pip`, such as FastAPI.

Using existing packages and modules is nice but writing your own is even better. In Python, a **module** is a single file containing declarations but can also contain instructions that will be executed when the module is first imported. You'll find in the following example the definition of a very simple module:

`chapter2_basics_module.py`

```
def module_function():  
    return "Hello world"  
  
print("Module is loaded")
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_module.py

This module only contains a function, `module_function`, and a `print` statement. Create a file containing this code at the root of your project directory and name it `module.py`. Then, open a Python interpreter and run this command:

```
>>> import module  
Module is loaded
```

Notice that the `print` statement was executed when you imported it. You can now use the following function:

```
>>> module.module_function()  
'Hello world'
```

Congratulations! You've just written your first Python module!

Now, let's see how to structure a **package**. A package is a way to organize modules in a hierarchy that you can then import using its namespace.

At the root of your project, create a directory named `package`. Inside, create another directory named `subpackage` and move `module.py` into it. Your project structure should look like the one shown in *Figure 2.1*:

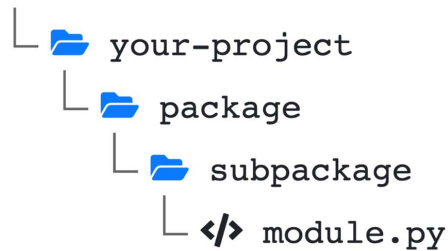


Figure 2.1 – Python package sample hierarchy

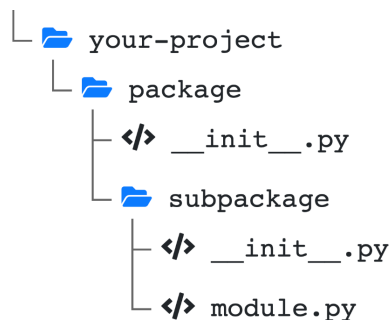
You can then import your module using the full namespace:

```
>>> import package.subpackage.module
```

```
Module is loaded
```

It works! However, to define a proper Python package, it's *strongly recommended* to create an empty `__init__.py` file at the root of each package and sub-package. In older Python versions, it was compulsory to make a package recognizable by the interpreter. This became optional in more recent versions, but there are actually some subtle differences between a package with an `__init__.py` file (a package) and one without (a **namespace package**). We won't explain it further in this book, but you could check the documentation about namespace packages here if you wish for more details: <https://packaging.python.org/guides/packaging-namespaces/>.

Therefore, you generally should always create `__init__.py` files. In our example, our project structure would finally look like this:

Figure 2.2 – Python package hierarchy with `__init__.py` files

It's worth noting that even if empty `__init__.py` files are perfectly fine, you can actually write some code in them. In this case, it is executed the first time you import the package or one of its sub-modules. It's useful to perform some initialization logic for your package. You now have a good overview of how to write some Python code. Feel free to write some small scripts to get acquainted with its peculiar syntax. We'll now explore more advanced topics about the language that will prove useful during our journey with FastAPI.

Operating over sequences – list comprehensions and generators

In this section, we'll cover what are probably the most idiomatic constructions in Python: list comprehensions and generators. You'll see that they are very useful for reading and transforming sequences of data with very minimal syntax.

List comprehensions

In programming, a very common task is to transform a sequence (let's say a *list*) into another, for example, to filter out or transform elements. Usually, you would write an operation as we did in one of the previous examples of this chapter:

chapter2_basics_02.py

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even = []

for number in numbers:
    if number % 2 == 0:
        even.append(number)

print(even) # [2, 4, 6, 8, 10]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_basics_02.py

With this approach, we simply iterate over each element, check a condition, and add the element in an accumulator if it passes this condition.

To go further into its readability philosophy, Python supports a neat syntax to perform this operation in only one statement: **list comprehensions**. Let's see what our previous example looks like with this syntax:

chapter2_list_comprehensions_01.py

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even = [number for number in numbers if number % 2 == 0]
print(even) # [2, 4, 6, 8, 10]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_01.py

That's it! Basically, a list comprehension works by *packing* a `for` loop and wrapping it with square brackets. The element to add to the result list appears first, followed by the iteration. Optionally, we can add a condition, as we did here, to filter some elements of the list input.

Actually, the result element can be any valid Python expression. In the following example, we use the `randint` function of the `random` standard module to generate a list of random integers:

chapter2_list_comprehensions_02.py

```
from random import randint, seed

seed(10) # Set random seed to make examples reproducible
random_elements = [randint(1, 10) for i in range(5)]
print(random_elements) # [10, 1, 7, 8, 10]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_02.py

This syntax is widely used by Python programmers and you'll probably grow quite fond of it. The nice thing about this syntax is that it also works for *sets* and *dictionaries*. Quite simply, just replace the square brackets with curly braces to generate a set:

chapter2_list_comprehensions_03.py

```
from random import randint, seed

seed(10) # Set random seed to make examples reproducible
random_unique_elements = {randint(1, 10) for i in range(5)}
print(random_unique_elements) # {8, 1, 10, 7}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_03.py

To create a dictionary, specify both the key and the value separated by a colon:

chapter2_list_comprehensions_04.py

```
from random import randint, seed

seed(10) # Set random seed to make examples reproducible
random_dictionary = {i: randint(1, 10) for i in range(5)}
print(random_dictionary) # {0: 10, 1: 1, 2: 7, 3: 8, 4: 10}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_04.py

Generators

You might think that if you replace the square brackets with parentheses, you could obtain a tuple. Actually, you get a **generator** object. The main difference between generators and list comprehensions is that elements are generated *on demand* and not computed and stored all at once in memory. You could see a generator as a recipe to generate values.

As we said, a generator can be defined simply by using the same syntax as list comprehensions, with parentheses:

chapter2_list_comprehensions_05.py

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_generator = (number for number in numbers if number % 2 == 0)
even = list(even_generator)
even_bis = list(even_generator)

print(even) # [2, 4, 6, 8, 10]
print(even_bis) # []
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_05.py

In this example, we define `even_generator` to output the even numbers of the `numbers` list. Then, we call the `list` constructor with this generator and assign it to the variable named `even`. This constructor will exhaust the iterator passed in the parameter and build a proper list. We do it a second time and assign it to `even_bis`.

As you see, `even` is a list with all the even numbers. However, `even_bis` is an *empty* list. This simple example is here to show you that a generator can be used *only once*. Once all the values have been produced, it's over.

This can be useful because you can start to iterate on the generator, stop to do something else, and then resume iterating.

Another way to create generators is to define **generator functions**. In the following example, we'll define a generator function that outputs even numbers from 2 to the limit passed in the argument:

chapter2_list_comprehensions_06.py

```
def even_numbers(max):
    for i in range(2, max + 1):
        if i % 2 == 0:
            yield i
```



```
even = list(even_numbers(10))
print(even) # [2, 4, 6, 8, 10]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_06.py

As you see in this function, we use the `yield` keyword instead of `return`. When the interpreter reaches this statement, it *pauses* the function execution and *yields* the value to the generator consumer. When the main program asks for another value, the function is resumed in order to yield again.

This allows us to implement complex generators, even ones that will output different types of values over their course. Another interesting property of generator functions is that they allow us to execute some instructions when they have finished to generate values. Let's add a `print` statement at the end of the function we just reviewed:

chapter2_list_comprehensions_07.py

```
def even_numbers(max):
    for i in range(2, max + 1):
        if i % 2 == 0:
            yield i
    print("Generator exhausted")

even = list(even_numbers(10))
print(even)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_list_comprehensions_07.py

If you execute it in a Python interpreter, you'll get this output:

```
$ python chapter2_list_comprehensions_07.py
Generator exhausted
[2, 4, 6, 8, 10]
```

We get `Generator exhausted` in the output, which means that our code *after* the last `yield` statement is well executed.

This is especially useful when you want to perform some *cleanup operations* after your generator has been exhausted: close a connection, remove temporary files, and so on.

Writing object-oriented programs

As we said in the first section of this chapter, Python is a multi-paradigm language, and one among those paradigms is **object-oriented programming**. In this section, we'll review how you can define classes and how you can instantiate and use objects. You'll see that Python syntax is once again very lightweight.

Defining a class

Defining a class in Python is straightforward: use the `class` keyword, type the name of your class, and begin a new block. You can then define methods under it just like you would for regular functions. Let's review an example:

chapter2_classes_objects_01.py

```
class Greetings:
    def greet(self, name):
        return f"Hello, {name}"

c = Greetings()
print(c.greet("John")) # "Hello, John"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_01.py

Notice that the first argument of each method must be `self`, which is a reference to the current object instance (the equivalent of `this` in other languages).

To instantiate a class, simply call the class as you would for a function and assign it to a variable. You can then access the methods using dot notation.

Class and method naming

By convention, classes should be named using **camel case**: `MyWonderfulClass` but not `my_wonderful_class`. Methods should use snake case like regular functions.

Obviously, you can also set **class properties**. To do this, we'll implement the `__init__` method, whose goal is to initialize values:

chapter2_classes_objects_02.py

```
class Greetings:
    def __init__(self, default_name):
        self.default_name = default_name

    def greet(self, name=None):
        return f"Hello, {name if name else self.default_name}"

c = Greetings("Alan")
print(c.default_name) # "Alan"
print(c.greet()) # "Hello, Alan"
print(c.greet("John")) # "Hello, John"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_02.py

In this example, `__init__` allows us to set a `default_name` property, which will be used by the `greet` method if no name is provided in the argument. As you see, you can simply access this property through dot notation.

Be careful though: `__init__` is not a constructor. In typical object-oriented languages, a constructor is a method to actually create the object in memory. In Python, when `__init__` is called, the object is already created in memory (notice we have access to the `self` instance). Actually, there is a method to define the constructor, `__new__`, but it's rarely used in Python.

Private methods and properties

In Python, there is no such thing as *private* methods or properties. Everything will always be accessible from the outside. However, by convention, you can prefix your private methods and properties with an underscore to *suggest* that they should be considered as private: `_private_method`.

You now know the basics of object-oriented programming in Python! We'll now focus on magic methods, which will allow us to do clever things with objects.

Implementing magic methods

Magic methods are a set of predefined methods that bear a special meaning in the language. They are easy to recognize as they start and end with two underscores. Actually, we already saw one of those magic methods: `__init__`! Those methods are not called directly but are used by the interpreter when using other constructs such as standard functions or operators.

To understand how they are useful, we'll review the most used. Let's start with `__repr__` and `__str__`.

Object representations - `__repr__` and `__str__`

When you define a class, it's generally useful to be able to get a readable and clear string representation of an instance. For this purpose, Python provides two magic methods: `__repr__` and `__str__`. Let's see how they work on a class representing a temperature in either degrees Celsius or degrees Fahrenheit:

chapter2_classes_objects_03.py

```
class Temperature:
    def __init__(self, value, scale):
        self.value = value
        self.scale = scale

    def __repr__(self):
        return f"Temperature({self.value}, {self.scale!r})"

    def __str__(self):
        return f"Temperature is {self.value} °{self.scale}"

t = Temperature(25, "C")
print(repr(t)) # "Temperature(25, 'C')"
print(str(t)) # "Temperature is 25 °C"
print(t)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_03.py

If you run this example, you'll notice that `print(t)` prints the same thing as `print(str(t))`. Through `print`, the interpreter called the `__str__` method to get the string representation of our object. This is what `__str__` is for: giving a *nice string representation* of an object for the end user.

On the other hand, you saw that even if very similar, we implemented `__repr__` in a different way. The purpose of this method is to give an *internal representation* of the object that is unambiguous. By convention, this should give the exact statement that would allow us to recreate the very same object.

Now that we can represent temperatures with our class, what would happen if we tried to compare them?

Comparison methods - `__eq__`, `__gt__`, `__lt__`, and so on

Of course, comparing two temperatures with different units would lead to unexpected results. Fortunately, magic methods allow us to overload the default operators to perform meaningful comparisons. Let's expand our previous example:

chapter2_classes_objects_04.py

```
class Temperature:
    def __init__(self, value, scale):
        self.value = value
        self.scale = scale
        if scale == "C":
            self.value_kelvin = value + 273.15
        elif scale == "F":
            self.value_kelvin = (value - 32) * 5/9 + 273.15
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_04.py

In the `__init__` method, we convert the temperature value in Kelvin given the current scale. This will help us to make comparisons. Then, let's define `__eq__` and `__lt__`:

chapter2_classes_objects_04.py

```
def __eq__(self, other):
    return self.value_kelvin == other.value_kelvin
```

```
def __lt__(self, other):  
    return self.value_kelvin < other.value_kelvin
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_04.py

As you see, those methods simply accept another argument, which is the other object instance to compare with. We then just have to perform our comparison logic. By doing this, we can perform comparison just as we would for any variable:

chapter2_classes_objects_04.py

```
tc = Temperature(25, "C")  
tf = Temperature(77, "F")  
tf2 = Temperature(100, "F")  
print(tc == tf) # True  
print(tc < tf2) # True
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_04.py

That's it! If you wish to have all the comparison operators available, you should also implement all the other comparison magic methods: `__le__`, `__gt__`, and `__ge__`.

The type of the other instance is not guaranteed

In this example, we assumed that the `other` variable was also a `Temperature` object. In the real world, however, this is not guaranteed and developers could try to compare `Temperature` with another object, which would likely lead to errors or weird behaviors. To prevent this, you should check the type of the `other` variable using `isinstance` to ensure we handle `Temperature`, or raise a proper exception otherwise.

Operators - `__add__`, `__sub__`, `__mul__`, and so on

Similarly, you could also define what would happen when trying to add or multiply two `Temperature` objects. We won't go into much detail here as it works exactly the same as the comparison operators.

Callable object – `__call__`

The last magic method we'll review is `__call__`. This one is a bit special because it enables you to call your object instance like a *regular function*. Let's take an example:

`chapter2_classes_objects_05.py`

```
class Counter:
    def __init__(self):
        self.counter = 0

    def __call__(self, inc=1):
        self.counter += inc

c = Counter()
print(c.counter)  # 0
c()
print(c.counter)  # 1
c(10)
print(c.counter)  # 11
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_05.py

`__call__` can be defined like any other method, with any argument you wish. The only difference is how you call it: you just pass the argument directly on the object instance variable as you would do for a regular function.

This pattern can be useful if you want to define a function that maintains some kind of local state, as we did here in our example, or in cases where you need to provide a **callable** object but have to set some parameters. Actually, this is the use case we'll encounter when defining class dependencies for FastAPI.

As we saw, magic methods are an excellent way to implement operations for our custom classes and make them easy to use in a pure object-oriented way. We haven't covered every magic method available but you can find the complete list on the official documentation: <https://docs.python.org/3/reference/datamodel.html#special-method-names>.

We'll now focus on another essential characteristic of object-oriented programming: inheritance.

Reusing logic and avoiding repetition with inheritance

Inheritance is one of the core concepts of object-oriented programming: it allows you to derive a new class from existing ones, enabling you to reuse some logic and overload the parts that are specific to this new one. Of course, this is supported in Python. We'll take very simple examples to understand the mechanism underneath.

First of all, let's take an example of a very simple inheritance:

chapter2_classes_objects_06.py

```
class A:
    def f(self):
        return "A"

class Child(A):
    pass
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_06.py

The `Child` class inherits from the `A` class. The syntax is simple: the class we want to inherit from is specified between parentheses after the child class name.

The `pass` statement

`pass` is a statement that *does nothing*. Since Python relies only on indentation to denote blocks, it's a useful statement to create an *empty block*, as you would do with curly braces in other programming languages.

In this example, we don't want to add some logic to the `Child` class, so we just write `pass`.

Another way to do it is to add a docstring just below the class definition.

If you wish to overload a method but still want to get the result of the parent method, you can call the `super` function:

chapter2_classes_objects_07.py

```
class A:
    def f(self):
        return "A"

class Child(A):
    def f(self):
        parent_result = super().f()
        return f"Child {parent_result}"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_07.py

You now know how to create basic inheritance in Python. But there is more: we can also have multiple inheritance!

Multiple inheritance

As its name suggests, multiple inheritance allows you to derive a child class from multiple classes. This way, you can combine the logic of several classes into one. Let's take an example:

chapter2_classes_objects_08.py

```
class A:
    def f(self):
        return "A"

class B:
    def g(self):
        return "B"

class Child(A, B):
    pass
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_08.py

Once again, the syntax is quite straightforward: just list all the parent classes with a comma. Now, the `Child` class can call both the `f` and `g` methods.

Mixins

Mixins are a common pattern in Python that take advantage of the multiple inheritance feature. Basically, mixins are short classes containing a single feature that you often want to reuse. You can then compose concrete classes by combining the mixins.

However, what would happen if both `A` and `B` classes implemented a method named `f`? Let's try it out:

chapter2_classes_objects_09.py

```
class A:
    def f(self):
        return "A"

class B:
    def f(self):
        return "B"

class Child(A, B):
    pass
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_classes_objects_09.py

If you call method `f` of `Child`, you'll get the value `"A"`. In this simple case, Python will consider the first matching method following the order of the parent classes. However, for more complex hierarchies, the resolution may not be so obvious: this is the purpose of the **Method Resolution Order (MRO)** algorithm. We won't go into much detail here but just know that it follows the C3 linearization principles. If you wish to know more, you can have a look at the official document explaining the algorithm implemented in Python: <https://www.python.org/download/releases/2.3/mro/>.

If you are confused about the MRO of your class, you can call the `mro` method on your class to get a list of considered classes in order:

```
>>> Child.mro()
[<class 'chapter2.chapter2_classes_objects_08.Child'>, <class
'chapter2.chapter2_classes_objects_08.A'>, <class 'chapter2.
chapter2_classes_objects_08.B'>, <class 'object'>]
```

Well done! You now have a good overview of object-oriented programming in Python. Those concepts will be helpful when defining dependencies in FastAPI.

We'll now review some of the most recent and trending features in Python upon which FastAPI relies heavily. We'll start with **type hinting**.

Type hinting and type checking with mypy

In the first section of this chapter, we said that Python was a dynamically typed language: the interpreter doesn't check types at compile time but rather at runtime. This makes the language a bit more flexible and the developer a bit more efficient. However, if you are experienced with that kind of language, you probably know that it's easy to produce errors and bugs in this context: forgetting arguments and type mismatch.

This is why Python introduced type hinting starting with *version 3.5*. The goal is to provide a syntax to annotate the source code with **type annotations**: each variable, function, and class can be annotated to give indications about the types they expect. This *doesn't mean* that Python becomes a statically typed language. Those annotations remain completely *optional* and are *ignored* by the interpreter. However, those annotations can be used by **static-type checkers**, which will check whether your code is valid and consistent following the annotations. Hence, it greatly helps you to reduce errors and write self-explanatory code. One of those tools, `mypy`, is widely used by the community in this context.

Getting started

To understand how type annotations work, we'll review a simple annotated function:

chapter2_type_hints_01.py

```
def greeting(name: str) -> str:
    return f"Hello, {name}"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_01.py

As you see here, we simply added the type of the name argument after a colon. We also specified the **return type** after an arrow. For built-in types, such as `str` or `int`, we can simply use them as type annotations. We'll see a little further in this section how to annotate more complex types such as lists or dictionaries.

We'll now install `mypy` to perform a type check on this file. This can be done like any other Python package:

```
$ pip install mypy
```

Then, you can run a type check on your source file:

```
$ mypy chapter2_type_hints_01.py  
Success: no issues found in 1 source file
```

As you see, `mypy` tells us that everything is good with our typing. Let's try to modify our code a bit to provoke a type error:

```
def greeting(name: str) -> int:  
    return f"Hello, {name}"
```

Quite simply, we just said that the return type of our function is now `int`, but we are still returning a string. If you run this code, it'll execute perfectly well: as we said, the interpreter ignores type annotations. However, let's see what `mypy` tells us about it:

```
$ mypy chapter2_type_hints_01.py  
chapter2/chapter2_type_hints_01.py:2: error: Incompatible  
return value type (got "str", expected "int")  
Found 1 error in 1 file (checked 1 source file)
```

This time, it complains. It clearly tells us what is wrong here: the return value is a string, while an integer was expected!

Code editors and IDE integration

Having type checking is good, but it may be a bit tedious to run `mypy` manually in the command line. Fortunately, it integrates well with the most popular code editors and IDEs. Once configured, it'll perform type checking while you type and show you the errors directly on the faulty lines. Type annotations also help the IDE to perform clever things such as *auto-completion*.

You can check on the official documentation of `mypy` how to set it up for your favorite editor: <https://github.com/python/mypy#ide-linter-integrations-and-pre-commit>.

You understand the basics of type hinting in Python. We'll now review more advanced examples, especially with non-scalar types.

The typing module

So far, we've seen how to annotate variables for scalar types such as `str` or `int`. But we've seen that there are data structures such as lists and dictionaries that are widely used in Python. For those and other types of utilities, Python introduced the `typing` module.

In the following example, we'll show how to type hint basic data structures in Python:

chapter2_type_hints_02.py

```
from typing import Dict, List, Set, Tuple

l: List[int] = [1, 2, 3, 4, 5]
t: Tuple[int, str, float] = (1, "hello", 3.14)
s: Set[int] = {1, 2, 3, 4, 5}
d: Dict[str, int] = {"a": 1, "b": 2, "c": 3}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_02.py

The `typing` module contains classes for type hinting lists, tuples, sets, and dictionaries. You simply have to import it and use it in your annotations. In this case, those classes expect you to provide the type of the values composing your structure. It's the same as the well-known concept of **generics** in object-oriented programming. In Python, they are defined using square brackets.

Built-in type annotations have changed in Python 3.9

Starting with Python 3.9, the method to annotate lists, tuples, sets, and dictionaries shown here is deprecated. This newer version of Python actually supports type hinting with a regular class, without the need to import another version from typing: `l: list[int] = [1, 2, 3, 4, 5]`.

Besides generic types, the `typing` module contains other utilities to cover more complex annotations. For example, having a list with elements of different types is perfectly valid in Python. To make this work with type checkers, we'll use the `Union` type:

chapter2_type_hints_03.py

```
from typing import List, Union

l: List[Union[int, float]] = [1, 2.5, 3.14, 5]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_03.py

`Union` is also a generic type accepting any number of types. In this case, our list will accept either integers or floating-point numbers. Of course, `mypy` will complain if you try to add an element to this list that is neither an `int` nor a `float` value.

There is also another case where this is useful. Quite often, you'll have function arguments or return types that return either a value or `None`. Thus, you could write something like this:

chapter2_type_hints_04.py

```
from typing import Union

def greeting(name: Union[str, None] = None) -> str:
    return f"Hello, {name if name else 'Anonymous'}"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_04.py

The allowed value is either a string or None. This works perfectly. However, this case is so common that `typing` provides a shortcut for this: `Optional`. So, you can write the following:

chapter2_type_hints_05.py

```
from typing import Optional
```

```
def greeting(name: Optional[str] = None) -> str:  
    return f"Hello, {name if name else 'Anonymous'}"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_05.py

When dealing with complex types, it may be useful to *alias* them and reuse them at will without the need to rewrite them each time. To do this, you can simply assign them as you would do for any variable:

chapter2_type_hints_06.py

```
from typing import Tuple
```

```
IntStringFloatTuple = Tuple[int, str, float]
```

```
t: IntStringFloatTuple = (1, "hello", 3.14)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_06.py

By convention, types should be named using camel case, like classes. Talking about classes, let's see how type hinting works with them:

chapter2_type_hints_07.py

```
from typing import List
```

```
class Post:
```

```
    def __init__(self, title: str) -> None:
```

```
self.title = title

def __str__(self) -> str:
    return self.title

posts: List[Post] = [Post("Post A"), Post("Post B")]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_07.py

Actually, there is nothing special about class type hinting. You just annotate the methods as you would for a regular function. If you need to use your class in an annotation, like here for a list of posts, you just have to use the class name.

Sometimes, you'll have to write a function or method that accepts another function in an argument. In this case, you'll need to give the **type signature** of this function. This is what the `Callable` class is for.

Type function signatures with Callable

It can be useful to have types for function signatures, especially when you need to pass functions as arguments of other functions. For this task, the `typing` module provides the `Callable` class. In the following example, we'll implement a function called `filter_list` that expects argument a list of integers and a function that returns a Boolean given an integer:

chapter2_type_hints_08.py

```
from typing import Callable, List

ConditionFunction = Callable[[int], bool]

def filter_list(l: List[int], condition: ConditionFunction) ->
List[int]:
    return [i for i in l if condition(i)]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_08.py

You see here that we define a `ConditionFunction` type alias thanks to `Callable`. Once again, this is a generic class that expects two things: first, the list of argument types, and then the return type. Here, we expect a single integer argument and the return type is a `Boolean`.

We can then use this type in the annotation of the `filter_list` function. `mypy` will then ensure that the condition function passed in the argument conforms to this signature. For example, we could write a simple function to check the parity of an integer, as shown in the next sample:

chapter2_type_hints_08.py

```
def is_even(i: int) -> bool:
    return i % 2 == 0

filter_list([1, 2, 3, 4, 5], is_even)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_08.py

It's worth noting, however, that there is no syntax to indicate optional or keyword arguments. In this case, you can write `Callable[... , bool]`, the ellipsis, `...`, meaning here *any number of arguments*.

Any and cast

In some situations, the code is so dynamic or complicated that it won't be possible to annotate it correctly or the type checker may not correctly infer the type. To help with this, the `typing` module provides two utilities: `Any` and `cast`.

The first one is a type annotation that tells the type checker that the variable or argument can be anything. In this case, any type of value will be valid for the type checker:

chapter2_type_hints_09.py

```
from typing import Any

def f(x: Any) -> Any:
    return x
```

```
f("a")
f(10)
f([1, 2, 3])
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_09.py

The second one, `cast`, is a function that lets you override the type inferred by the type checker. It'll force the type checker to consider the type you specify:

chapter2_type_hints_10.py

```
from typing import Any, cast

def f(x: Any) -> Any:
    return x

a = f("a") # inferred type is "Any"
a = cast(str, f("a")) # forced type to be "str"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_type_hints_10.py

Be careful though: the `cast` function is only meaningful for type checkers. As for every other type of annotation, the interpreter completely ignores it and *doesn't* perform a real cast.

While convenient, try to refrain from using those utilities too often. If everything is `Any` or casted to a different type, you completely miss the benefits of static type checking.

As we saw, type hinting and type checking are really helpful to help reduce errors while developing and keep high-quality code. But that's not all. Actually, Python allows you to retrieve type annotations *at runtime* and perform some logic based on them. This enables you to do clever things such as **dependency injection**: just by type hinting an argument in a function, a library can automatically interpret it and inject the corresponding value at runtime. This concept is at the heart of FastAPI.

Another key approach in FastAPI is **asynchronous I/O**. This will be the last subject we'll cover in this chapter.

Asynchronous I/O

If you have already worked with JavaScript and Node.js, you have probably come across the concepts of *promises* and the `async/await` keywords, which are characteristic of the asynchronous I/O paradigm. Basically, this is a way to make I/O operations non-blocking and allow the program to perform other tasks while the read or write operation is ongoing. The main motivation behind this is that I/O operations are *slow*: reading from disk, network requests are a *million* times slower than reading from RAM or processing instructions. In the following example, we have a simple script that reads a file on disk:

chapter2_asyncio_01.py

```
with open(__file__) as f:
    data = f.read()
# The program will block here until the data has been read
print(data)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_asyncio_01.py

We see that the script will block until we have retrieved the data from the disk and, as we said, this can be long. 99% percent of the execution time of the program is spent on waiting for the disk. Usually, it's not an issue for simple scripts like this because you probably don't have to perform other operations in the meantime.

However, in other situations, there could have been the opportunity to perform other tasks. The typical case that is of great interest in this book is web servers. Imagine we have a first user that makes a request performing a 10-seconds-long database query before sending the response. If a second user makes another request in the meantime, they'll have to wait for the first response to finish before getting their answer.

To solve this, traditional Python web servers based on the **Web Server Gateway Interface (WSGI)**, such as Flask or Django, spawn several **workers**. Those are sub-processes of the web server that are all able to answer requests. If one is busy processing a long request, others can answer new coming requests.

With asynchronous I/O, a single process won't block when processing a request with a long I/O operation. While it waits for this operation to finish, it can answer other requests. When the I/O operation is done, it resumes the request logic and can finally answer the request.

Technically, this is achieved through the concept of an **event loop**. Think of it as a conductor that will manage all the asynchronous tasks you'll send to it. When data is available or when the write operation is done for one of those tasks, it'll ping the main program so that it can perform the next operations. Underneath, it relies upon the operating system `select` and `poll` calls, which are precisely there to ask for events about I/O operations at an operating system level. You can read very interesting details about this in the article *Async IO on Linux: select, poll, and epoll* by Julia Evans: <https://jvns.ca/blog/2017/06/03/async-io-on-linux--select--poll--and-epoll/>.

Python first implemented asynchronous I/O in version 3.4 and has since greatly evolved, notably with the introduction of the `async/await` keywords in version 3.6. All the utilities to manage this paradigm are available through the standard `asyncio` module. Not long after, the spiritual successor of WSGI for asynchronous-enabled web servers, **Asynchronous Server Gateway Interface (ASGI)**, was introduced. FastAPI relies on this, and this is one of the reasons why it shows such great *performance*.

We'll now review the basics of asynchronous programming in Python. The following example is a simple *Hello world* using `asyncio`:

chapter2_asyncio_02.py

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_asyncio_02.py

When you wish to define an asynchronous function, you just have to add the `async` keyword before `def`. This allows you to use the `await` keyword inside it. Such `async` functions are called **coroutines**.

Inside it, we perform a first `print` statement and then call the `asyncio.sleep` coroutine. This is the `async` equivalent of `time.sleep` that blocks the program for a given number of seconds. Notice that we prefixed the call with the `await` keyword. This means that we want to wait for this coroutine to finish before proceeding. This is the main benefit of `async/await` keywords: writing code that looks like synchronous code. If we omitted `await`, the coroutine object would have been created but never executed.

Finally, notice that we use the `asyncio.run` function. This is the machinery that will create a new event loop, execute your coroutine, and return its result. It should be the main entry point of your `async` program.

This example is nice but not very interesting from an asynchronous point of view; since we are waiting for only one operation, this is not very impressive. Let's see an example where we execute two coroutines concurrently:

chapter2_asyncio_03.py

```
import asyncio

async def printer(name: str, times: int) -> None:
    for i in range(times):
        print(name)
        await asyncio.sleep(1)

async def main():
    await asyncio.gather(
        printer("A", 3),
        printer("B", 3),
    )

asyncio.run(main())
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter2/chapter2_asyncio_03.py

Here, we have a `printer` coroutine that prints its name a given number of times. Between each print, it sleeps for 1 second.

Then, our main coroutine uses the `asyncio.gather` utility, which schedules several coroutines for concurrent execution. If you run this script, you'll get the following result:

```
$ python chapter2_asyncio_03.py
A
B
A
B
A
B
```

A succession of A and B means that our coroutines were executed concurrently and that we didn't wait for the first one to finish before starting the second one.

You might wonder why we added the `asyncio.sleep` call in this example. Actually, if we removed it, we would have obtained this result:

```
A
A
A
B
B
B
```

That doesn't look very concurrent, and indeed it's not. This is one of the main pitfalls of `asyncio`: writing code in a coroutine *doesn't necessarily mean* that it won't block. Regular operations such as computations are blocking and will block the event loop. Usually, this is not a problem since those operations are fast. The only operations that won't block are proper I/O operations that are *designed* to work asynchronously. This is different from **multiprocessing** where operations are executed on child processes, which, by nature, doesn't block the main one.

Because of this, you'll have to be careful when choosing a third-party library for interacting with databases, APIs, and so on. Some have been adapted to work asynchronously, or some alternatives have been developed in parallel to the standard ones. We'll see some of them in the next chapters, especially when working with databases.

We'll end this quick introduction to asynchronous I/O here. There are some other subtleties underneath but, generally, the basics we've seen here will allow you to leverage the power of `asyncio` with FastAPI.

Summary

Congratulations! Through this chapter, you've discovered the basics of the Python language, a very clean and efficient language to work with. You've then been introduced to the more advanced concepts of list comprehensions and generators, which are idiomatic ways of handling sequences of data. Python is also a multi-paradigm language and you've seen how to leverage the object-oriented syntax.

Finally, you've discovered some of the most recent features of the language: type hinting, which allows static-type checking to reduce errors and speed up development, and asynchronous I/O, a set of new tools and syntax to maximize performance and allow concurrency while doing I/O-bound operations.

You're now ready to begin your journey with FastAPI! You'll see that the framework takes advantage of all those Python features to propose a fast and enjoyable development experience. In the next chapter, you'll learn how to write your very first REST API with FastAPI.

3

Developing a RESTful API with FastAPI

Now it's time to begin learning about **FastAPI**! In this chapter, we'll cover the basics of FastAPI. We'll go through very simple and focused examples that will demonstrate the different features of FastAPI. Each example will lead to a working API endpoint that you'll be able to test yourself using HTTPie. In the final section of this chapter, we'll show you a more complex FastAPI project, with routes split across several files. It will give you an overview of how you can structure your own application.

By the end of this chapter, you'll know how to start a FastAPI application and how to write an API endpoint. You'll also be able to handle request data and build a response according to your own logic. Finally, you'll learn a way to structure a FastAPI project into several modules that will be easier to maintain and work with in the long term.

In this chapter, we'll cover the following main topics:

- Creating the first endpoint and running it locally
- Handling request parameters
- Customizing the response
- Structuring a bigger project with multiple routers

Technical requirements

For the examples in this chapter, you'll require a Python virtual environment, just as we set up in *Chapter 1, Python Development Environment Setup*.

You can find all the code examples of this chapter in the dedicated GitHub repository at <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter3>.

Creating the first endpoint and running it locally

FastAPI is a framework that aims at being easy to use and quick to write. In the following example, you'll realize that this is not just a promise. In fact, creating an API endpoint involves just a few lines:

chapter3_first_endpoint_01.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def hello_world():
    return {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_first_endpoint_01.py

In this example, we define a GET endpoint at the root path, which always returns the `{"hello": "world"}` JSON response. To do this, we first instantiate a FastAPI object, `app`. This will be the main application object that will wire all of the API routes.

Then, we simply define a coroutine that contains our route logic, the **path operation function**. Its return value is automatically handled by FastAPI to produce a proper HTTP response with a JSON payload.

Here, the most important part of the code is probably the line starting with `@`, which can be found above the coroutine definition, the **decorator**. In Python, a decorator is syntactic sugar that allows you to wrap a function or class with common logic without compromising readability. It's roughly equivalent to `app.get("/") (hello_world)`.

FastAPI exposes *one decorator per HTTP method* to add new routes to the application. The one that is shown here adds a `GET` endpoint with the path as the first argument.

Now, let's run this API. Copy the example to the root of your project and run the following command:

```
$ uvicorn chapter3_first_endpoint_01:app
INFO:      Started server process [13300]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press
CTRL+C to quit)
```

As we mentioned in the *Asynchronous I/O* section of *Chapter 2, Python Programming Specificities*, FastAPI exposes an ASGI-compatible application. To run it, we require a web server compatible with this protocol. Uvicorn is a good option to use. It gives a command to quickly start a web server. In the first argument, it expects the *dotted namespace* of the Python module, which contains your app instance, followed by a colon, `:`, and, finally, the variable name of your ASGI app instance (in our example, this is `app`). Afterward, it takes care of instantiating the application and exposing it on your local machine.

Let's try our endpoint with HTTPie:

```
$ http http://localhost:8000
HTTP/1.1 200 OK
content-length: 17
content-type: application/json
date: Tue, 23 Mar 2021 07:35:16 GMT
server: uvicorn

{
  "hello": "world"
}
```

It works! As you can see, we did get a JSON response with the payload that we wanted, using just a few lines of Python and a command!

One of the most beloved features of FastAPI is the *automatic interactive documentation*. If you open `http://localhost:8000/docs` in your browser, you should get a web interface that looks similar to the following screenshot:

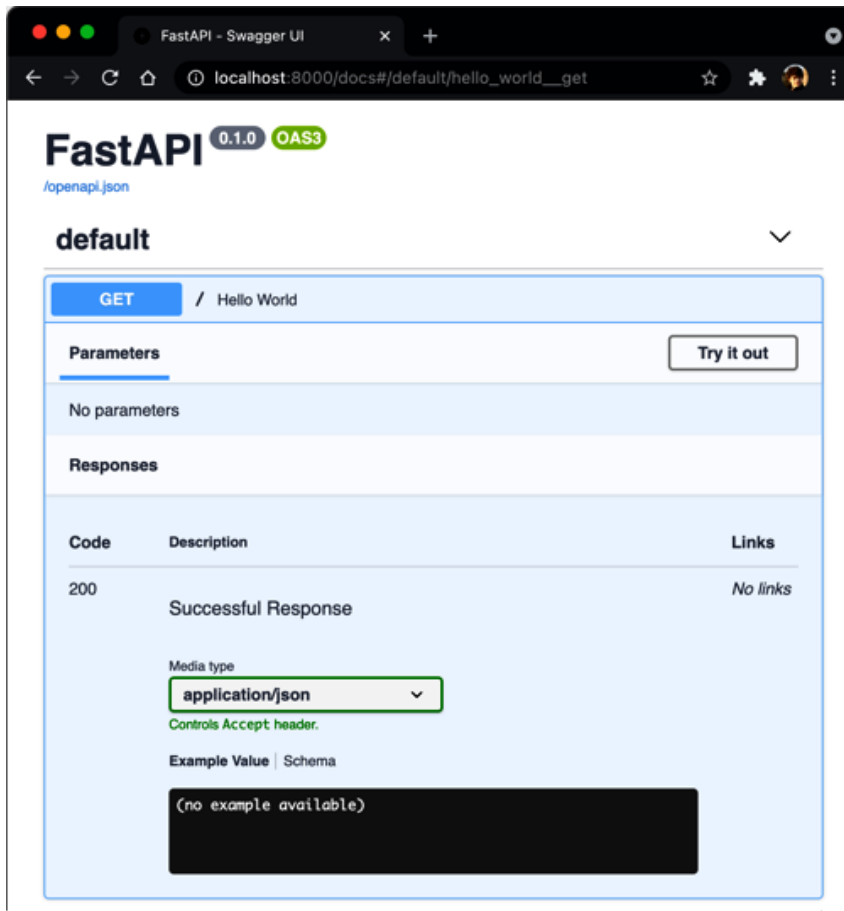


Figure 3.1 – The FastAPI automatic interactive documentation

FastAPI will automatically list all of your defined endpoints and provide documentation about the expected inputs and outputs. You can even try each endpoint directly in this web interface. Under the hood, it relies on the OpenAPI Specification and the associated tools by Swagger. You can read more about this on their official website at `https://swagger.io/`.

That's it! You've created your very first API with FastAPI. Of course, this is just a very simple example, but next, you'll learn how to handle input data and start making meaningful things!

On the shoulders of giants

It's worth noting that FastAPI is built upon two main Python libraries: Starlette, a low-level ASGI web framework (<https://www.starlette.io/>), and pydantic, a data validation library that is based on type hints (<https://pydantic-docs.helpmanual.io/>).

Handling request parameters

The main goal of a REST API is to provide a structured way in which to interact with data. As such, it's crucial for the end user to send some information to tailor the response they need, such as path parameters, query parameters, body payloads, or headers.

To handle them, usually, web frameworks ask you to manipulate a request object to retrieve the parts you are interested in and manually apply validation. However, that's not necessary with FastAPI! Indeed, it allows you to define all of your parameters declaratively. Then, it'll automatically retrieve them in the request and apply validations based on the type hints. This is why we introduced type hinting in *Chapter 2, Python Programming Specificities*; it's used by FastAPI to perform data validation!

Next, we'll explore how you can use this feature to retrieve and validate this input data from different parts of the request.

Path parameters

The API path is the main thing that the end user will interact with. Therefore, it's a good spot for dynamic parameters. A typical example is to put the unique identifier of an object we want to retrieve, such as `/users/123`. Let's examine how to define this with FastAPI:

chapter3_path_parameters_01.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{id}")
async def get_user(id: int):
    return {"id": id}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_01.py

In this example, we defined an API that expects an integer in the last part of its path. We did this by putting the parameter name in the path around curly braces. Then, we defined this same parameter as an argument for our path operation function. Notice that we add a type hint to specify the parameter is an integer.

Let's run this example. You can refer to the previous section, *Creating the first endpoint and running it locally*, to learn how to run a FastAPI app with `uvicorn`.

First, we'll try to make a request by omitting our path parameter:

```
$ http http://localhost:8000/users
HTTP/1.1 404 Not Found
content-length: 22
content-type: application/json
date: Wed, 24 Mar 2021 07:23:47 GMT
server: uvicorn

{
  "detail": "Not Found"
}
```

We get a response with a 404 status. That's expected: our route awaits a parameter after `/users`, so if we omit it, it simply doesn't match any pattern.

Now, let's try it using a proper integer parameter:

```
$ http http://localhost:8000/users/123
HTTP/1.1 200 OK
content-length: 10
content-type: application/json
date: Wed, 24 Mar 2021 07:26:23 GMT
server: uvicorn

{
  "id": 123
}
```

It works! We get a 200 status, and the response does contain the integer we passed in the parameter. Notice that it has been properly *cast* as an integer.

So, what happens if we pass a value that's not a valid integer? Let's find out:

```
$ http http://localhost:8000/users/abc
HTTP/1.1 422 Unprocessable Entity
content-length: 99
content-type: application/json
date: Wed, 24 Mar 2021 07:28:11 GMT
server: uvicorn

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

We get a response with a 422 status! Since `abc` is not a valid integer, the validation fails and outputs an error. Notice that we have a very detailed and structured error response telling us exactly which element caused the error and why. All we need to do to trigger this validation is to *type hint* our parameter!

Of course, you are not limited to just one path parameter. You can have as many as you want, with different types. In the following example, we've added a `type` parameter of the string type:

chapter3_path_parameters_02.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{type}/{id}/")
async def get_user(type: str, id: int):
    return {"type": type, "id": id}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_02.py

This works well, but the endpoint will accept any string as the `type` parameter.

Limiting allowed values

So, what if we just want to accept a limited set of values? Once again, we'll lean on type hinting. Python has a very useful class for this: `Enum`. An enumeration is a way to list all the valid values for a specific kind of data. Let's define an `Enum` class that will list the different types of users:

chapter3_path_parameters_03.py

```
from enum import Enum

from fastapi import FastAPI

class UserType(str, Enum):
    STANDARD = "standard"
    ADMIN = "admin"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_03.py

To define a string enumeration, we inherit from both the `str` type and the `Enum` class. Then, we simply list the allowed values as class properties: the property name and its actual string value. Finally, we need to type hint the `type` argument using this class:

chapter3_path_parameters_03.py

```
@app.get("/users/{type}/{id}/")
async def get_user(type: UserType, id: int):
    return {"type": type, "id": id}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_03.py

If you run this example and call the endpoint with a type that is not in the enumeration, you'll get the following response:

```
$ http http://localhost:8000/users/hello/123/
HTTP/1.1 422 Unprocessable Entity
content-length: 184
content-type: application/json
date: Thu, 25 Mar 2021 06:30:36 GMT
server: uvicorn

{
  "detail": [
    {
      "ctx": {
        "enum_values": [
          "standard",
          "admin"
        ]
      },
      "loc": [
        "path",
        "type"
      ],
      "msg": "value is not a valid enumeration member;"
```



```
permitted: 'standard', 'admin',  
          "type": "type_error.enum"  
        }  
    ]  
}
```

As you can see, you get a nice validation error, with the allowed values for this parameter!

Advanced validation

We can take one step further by defining more advanced validation rules, particularly for numbers and strings. In this case, the type of hint is no longer enough. We'll rely on the functions provided by FastAPI, allowing us to set some options on each of our parameters. For path parameters, the function is named `Path`. In the following example, we'll only allow an `id` argument that is greater than or equal to 1:

chapter3_path_parameters_04.py

```
from fastapi import FastAPI, Path  
  
app = FastAPI()  
  
@app.get("/users/{id}")  
async def get_user(id: int = Path(..., ge=1)):  
    return {"id": id}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_04.py

There are several things to pay attention to here: the result of `Path` is used as a *default value* for the `id` argument in the path operation function.

Additionally, you can see that we use the **ellipsis** syntax as the first parameter of `Path`. Indeed, it expects the default value for the parameter as the first argument. In this scenario, we don't want a default value: the parameter is required. Therefore, ellipses are here to tell FastAPI that we don't want a default value.

Then, we can add the keyword arguments that we are interested in. In our example, this is `ge`, greater than or equal to, and its associated value. There are a number of validations available, as follows:

- `gt`: Greater than
- `ge`: Greater than or equal to
- `lt`: Less than
- `le`: Less than or equal to

There are also validation options for string values, which are based on the *length* and the use of a *regular expression*. In the following example, we want to define a path parameter that accepts license plates in the form of *AB-123-CD* (French license plates). The first approach would be to force the string to be of length 9 (that is, two letters, a dash, three digits, a dash, and two letters):

chapter3_path_parameters_05.py

```
@app.get("/license-plates/{license}")
async def get_license_plate(license: str = Path(..., min_
length=9, max_length=9)):
    return {"license": license}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_05.py

Now we just have to define the `min_length` and `max_length` keyword arguments, just as we did for the number of validations. Of course, a better solution for this use case is to use a regular expression to validate the license plate number:

chapter3_path_parameters_06.py

```
@app.get("/license-plates/{license}")
async def get_license_plate(license: str = Path(..., regex=r"^\\
w{2}-\\d{3}-\\w{2}$")):
    return {"license": license}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_path_parameters_06.py

Thanks to this regular expression, we only accept strings that exactly match the license plate format. Notice that the regular expression is prefixed with `r`. Just like `f-strings`, this is a Python syntax that is used to indicate that the following string should be considered as a regular expression.

Parameter metadata

Data validation is not the only option accepted by the parameter function. You can also set options that will add information about the parameter in the automatic documentation, such as `title`, `description`, and `deprecated`.

Now you should be able to define path parameters and apply some validation to them. Other useful parameters to put inside the URL are **query parameters**. We'll discuss them next.

Query parameters

Query parameters are a common way to add some dynamic parameters to a URL. You find them at the end of the URL in the following form: `?param1=foo¶m2=bar`. In a REST API, they are commonly used on read endpoints to apply pagination, a filter, a sorting order, or selecting fields.

You'll discover that they are quite straightforward to define with FastAPI. In fact, they use the exact same syntax as path parameters:

chapter3_query_parameters_01.py

```
@app.get("/users")
async def get_user(page: int = 1, size: int = 10):
    return {"page": page, "size": size}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_query_parameters_01.py

You simply have to declare them as arguments of your path operation function. If they don't appear in the path pattern, as they do for path parameters, FastAPI automatically considers them to be query parameters. Let's try it:

```
$ http "http://localhost:8000/users?page=5&size=50"
HTTP/1.1 200 OK
content-length: 20
content-type: application/json
date: Thu, 25 Mar 2021 07:17:01 GMT
server: uvicorn

{
  "page": 5,
  "size": 50
}
```

Here, you can see that we have defined a default value for those arguments, which means they are *optional* when calling the API. Of course, if you wish to define a *required* query parameter, simply leave out the default value:

chapter3_query_parameters_01.py

```
from enum import Enum
from fastapi import FastAPI

class UsersFormat(str, Enum):
    SHORT = "short"
    FULL = "full"

app = FastAPI()

@app.get("/users")
async def get_user(format: UsersFormat):
    return {"format": format}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_query_parameters_01.py

Now, if you omit the `format` parameter in the URL, you'll get a 422 error response. Additionally, notice that, in this example, we defined a `UsersFormat` enumeration to limit the number of allowed values for this parameter; this is exactly what we did in the previous section for path parameters.

We also have access to more advanced validations through the `Query` function. It works in the same way that we demonstrated in the *Path parameters* section:

chapter3_query_parameters_01.py

```
@app.get("/users")
async def get_user(page: int = Query(1, gt=0), size: int =
    Query(10, le=100)):
    return {"page": page, "size": size}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_query_parameters_01.py

Here, we force the page to be *greater than 0* and the size to be *less than or equal to 100*. Notice how the default parameter value is the first argument of the `Query` function.

Naturally, when it comes to sending request data, the most obvious way is to use the request body. Let's examine how it works with FastAPI next.

The request body

The body is the part of the HTTP request that contains raw data, representing documents, files, or form submissions. In a REST API, it's usually encoded in JSON and used to create structured objects in a database.

For the simplest cases, retrieving data from the body works exactly like query parameters. The only difference is that you always have to use the `Body` function; otherwise, FastAPI will look for it inside the query parameters by default. Let's explore a simple example where we want to post some user data:

chapter3_request_body_01.py

```
@app.post("/users")
async def create_user(name: str = Body(...), age: int =
    Body(...)):
    return {"name": name, "age": age}
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_request_body_01.py
```

In the same way as query parameters, we define each argument with a type hint along with the `Body` function with no default value to make them required. Let's try the following endpoint:

```
$ http -v POST http://localhost:8000/users name="John" age=30
POST /users HTTP/1.1
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 29
Content-Type: application/json
Host: localhost:8000
User-Agent: HTTPie/2.4.0

{
  "age": "30",
  "name": "John"
}

HTTP/1.1 200 OK
content-length: 24
content-type: application/json
date: Sun, 28 Mar 2021 08:17:26 GMT
server: uvicorn

{
  "age": 30,
  "name": "John"
}
```

Here, we've shown the request payload with the `-v` option so that you can clearly see the JSON payload we sent. FastAPI successfully retrieves the data for each field from the payload. If you send a request with a missing or invalid field, you'll get a 422 status error response.

You also have access to more advanced validation through the `Body` function. It works in the same way as we demonstrated in the *Path parameters* section.

However, defining payload validations like this has some major drawbacks. First, it's quite verbose and makes the path operation function prototype huge, especially for bigger models. Second, usually, you'll need to reuse the data structure on other endpoints or in other parts of your application.

This is why FastAPI uses **pydantic models** for data validation. Pydantic is a Python library for *data validation* and is based on classes and type hints. In fact, the `Path`, `Query`, and `Body` functions that we've learned about so far use pydantic under the hood!

By defining your own pydantic models and using them as type hints in your path arguments, FastAPI will automatically instantiate a model instance and validate the data. Let's rewrite our previous example using this method:

chapter3_request_body_02.py

```
from fastapi import FastAPI
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int

app = FastAPI()

@app.post("/users")
async def create_user(user: User):
    return user
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_request_body_02.py

First, we import `BaseModel` from `pydantic`. This is the base class that *every* model should inherit from. Then, we define our `User` class and simply list all of the properties as *class properties*. Each one of them should have a proper type hint: this is how `Pydantic` will be able to validate the type of the field.

Finally, we just declare `user` as an argument for our path operation function with the `User` class as a type hint. FastAPI automatically understands that the user data can be found in the request payload. Inside the function, you have access to a proper `User` object instance, where you can access individual properties by simply using the dot notation, such as `user.name`.

Notice that if you just return the object, FastAPI is smart enough to convert it automatically into JSON to produce the HTTP response.

In the following chapter, that is, *Chapter 4, Managing pydantic Data Models in FastAPI*, we'll explore, in more detail, the possibilities of pydantic, particularly in terms of validation.

Multiple objects

Sometimes, you might find that you have several objects that you wish to send in the same payload all at once. For example, both `user` and `company`. In this scenario, you can simply add several arguments that have been type hinted by a pydantic model, and FastAPI will automatically understand that there are several objects. In this configuration, it will expect a body containing each object *indexed by its argument name*:

chapter3_request_body_03.py

```
@app.post("/users")
async def create_user(user: User, company: Company):
    return {"user": user, "company": company}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_request_body_03.py

Here, `Company` is a simple pydantic model with a single string `name` property. In this configuration, FastAPI expects a payload that looks similar to the following:

```
{
  "user": {
    "name": "John",
    "age": 30
  },
  "company": {
    "name": "ACME"
  }
}
```


For more complex JSON structures, it's advised that you *pipe* a formatted JSON into HTTPie rather than use parameters. Let's try this as follows:

```
$ echo '{"user": {"name": "John", "age": 30}, "company": {"name": "ACME"}}' | http POST http://localhost:8000/users
HTTP/1.1 200 OK
content-length: 59
content-type: application/json
date: Sun, 28 Mar 2021 08:54:11 GMT
server: uvicorn

{
  "company": {
    "name": "ACME"
  },
  "user": {
    "age": 30,
    "name": "John"
  }
}
```

And that's it!

You can even add **singular body values** with the Body function, just as we saw at the beginning of this section. This is useful if you wish to have a single property that's not part of any model:

chapter3_request_body_04.py

```
@app.post("/users")
async def create_user(user: User, priority: int = Body(...,
ge=1, le=3)):
    return {"user": user, "priority": priority}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_request_body_04.py

The `priority` property is an integer between 1 and 3, which is expected *beside* the user object:

```
$ echo '{"user": {"name": "John", "age": 30}, "priority": 1}' |  
http POST http://localhost:8000/users  
HTTP/1.1 200 OK  
content-length: 46  
content-type: application/json  
date: Sun, 28 Mar 2021 09:02:59 GMT  
server: uvicorn  
  
{  
  "priority": 1,  
  "user": {  
    "age": 30,  
    "name": "John"  
  }  
}
```

You now have a good overview of how to handle JSON payload data. However, sometimes, you'll find that you need to accept more traditional-form data or even file uploads. Let's find out how to do this next!

Form data and file uploads

Even if REST APIs work most of the time with JSON, sometimes, you might have to handle form-encoded data or file uploads, which have been encoded either as `application/x-www-form-urlencoded` or `multipart/form-data`.

Once again, FastAPI allows you to implement this case very easily. However, you'll need an additional Python dependency, `python-multipart`, to handle this kind of data. As usual, you can install it with `pip`:

```
$ pip install python-multipart
```

Then, you can use the FastAPI features that are dedicated to form data. First, let's take a look at how you can handle simple form data.

Form data

The method to retrieve form data fields is similar to the one we discussed in the *The request body* section to retrieve singular JSON properties. The following example is roughly the same as the one you explored there. However, this example expects form-encoded data instead of JSON:

chapter3_form_data_01.py

```
@app.post("/users")
async def create_user(name: str = Form(...), age: int =
    Form(...)):
    return {"name": name, "age": age}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_form_data_01.py

The only difference here is that we use the `Form` function instead of `Body`. You can try this endpoint with `HTTPie` and the `--form` option to force the data to be form-encoded:

```
$ http -v --form POST http://localhost:8000/users name=John
age=30
POST /users HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 16
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Host: localhost:8000
User-Agent: HTTPie/2.4.0

name=John&age=30

HTTP/1.1 200 OK
content-length: 24
content-type: application/json
date: Sun, 28 Mar 2021 14:50:07 GMT
server: uvicorn
```

```
{  
  "age": 30,  
  "name": "John"  
}
```

Pay attention to how the `Content-Type` header and the body data representation have changed in the request. You can also see that the response is still provided in JSON. Unless specified otherwise, FastAPI will always output a JSON response by default, no matter the form of the input data.

Of course, the validation options we saw for `Path`, `Query`, and `Body` are still available. You can find a description for each of them in the *Path parameters* section.

It's worth noting that, contrary to JSON payloads, FastAPI doesn't allow you to define pydantic models to validate form data. Instead, you have to manually define each field as an argument for the path operation function.

Now, let's go on to discuss how to handle file uploads.

File uploads

Uploading files is a common requirement for web applications, whether this is images or documents. FastAPI provides a parameter function, `File`, that enables this.

Let's take a look at a simple example where you can directly retrieve a file as a bytes object:

Chapter3_file_uploads_01.py

```
from fastapi import FastAPI, File  
  
app = FastAPI()  
  
@app.post("/files")  
async def upload_file(file: bytes = File(...)):  
    return {"file_size": len(file)}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_file_uploads_01.py

Once again, you can see that the approach is still the same: we define an argument for the path operation function, `file`, we add a type of hint, `bytes`, and then we use the `File` function as a default value for this argument. By doing this, FastAPI understands that it will have to retrieve raw data in a part of the body named `file` and return it as `bytes`.

We simply return the size of this file by calling the `len` function on this `bytes` object.

In the code example repository, you should be able to find a picture of a cat: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/assets/cat.jpg>.

Let's upload it on our endpoint using HTTPie. To upload a file, type in the name of the file upload field (here, it is `file`), followed by `@` and the path of the file you want to upload. Don't forget to set the `--form` option:

```
$ http --form POST http://localhost:8000/files file@./assets/
cat.jpg
HTTP/1.1 200 OK
content-length: 19
content-type: application/json
date: Mon, 29 Mar 2021 06:42:02 GMT
server: uvicorn
{
  "file_size": 71457
}
```

It works! We have correctly got the size of the file in bytes.

One drawback to this approach is that the uploaded file is entirely stored *in memory*. So, while it'll work for small files, it is likely that you'll run into issues for larger files. Besides, manipulating a `bytes` object is not always convenient for file handling.

To fix this problem, FastAPI provides an `UploadFile` class. This class will store the data in memory up to a certain threshold and, after this, will automatically store it *on disk* in a temporary location. This allows you to accept much larger files without running out of memory. Furthermore, the exposed object instance exposes useful metadata, such as the content type, and a **file-like** interface. This means that you can manipulate it as a regular file in Python and that you can feed it to any function that expects a file.

To use it, you simply have to specify it as a type hint instead of `bytes`:

chapter3_file_uploads_02.py

```
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files")
async def upload_file(file: UploadFile = File(...)):
    return {"file_name": file.filename, "content_type": file.
           content_type}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_file_uploads_02.py

Notice that, here, we return the `filename` and `content_type` properties. The `content_type` is especially useful for *checking the type* of the uploaded file and possibly rejecting it if it's not one of the types that you expect.

Here is the result with HTTPie:

```
$ http --form POST http://localhost:8000/files file@./assets/
cat.jpg
HTTP/1.1 200 OK
content-length: 51
content-type: application/json
date: Mon, 29 Mar 2021 06:58:20 GMT
server: uvicorn

{
  "content_type": "image/jpeg",
  "file_name": "cat.jpg"
}
```

You can even accept multiple files by type hinting the argument as a list of `UploadFile`:

chapter3_file_uploads_03.py

```
@app.post("/files")
async def upload_multiple_files(files: List[UploadFile] =
File(...)):
    return [
        {"file_name": file.filename, "content_type": file.
content_type}
        for file in files
    ]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_file_uploads_03.py

To upload several files with HTTPie, simply repeat the argument. It should appear as follows:

```
$ http --form POST http://localhost:8000/files files@./assets/
cat.jpg files@./assets/cat.jpg
HTTP/1.1 200 OK
content-length: 105
content-type: application/json
date: Mon, 29 Mar 2021 12:52:45 GMT
server: uvicorn

[
  {
    "content_type": "image/jpeg",
    "file_name": "cat.jpg"
  },
  {
    "content_type": "image/jpeg",
    "file_name": "cat.jpg"
  }
]
```

Now, you should be able to handle form data and file uploads in a FastAPI application.

So far, you've learned how to manage user-facing data. However, there are also very interesting pieces of information that are less visible: **headers**. We'll explore them next.

Headers and cookies

Besides the URL and the body, another major part of the HTTP request are the headers. They contain all sorts of metadata that can be useful when handling requests. A common usage is to use them for authentication, for example, via the famous **cookies**.

Once again, retrieving them in FastAPI only involves a type hint and a parameter function. Let's take a look at a simple example where we want to retrieve a header named `Hello`:

chapter3_headers_cookies_01.py

```
@app.get("/")
async def get_header(hello: str = Header(...)):
    return {"hello": hello}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_headers_cookies_01.py

Here, you can see that we simply have to use the `Header` function as a default value for the `hello` argument. The name of the argument determines the *key of the header* that we want to retrieve. Let's see this in action:

```
$ http GET http://localhost:8000 'Hello: World'
HTTP/1.1 200 OK
content-length: 17
content-type: application/json
date: Mon, 29 Mar 2021 13:28:36 GMT
server: uvicorn

{
  «hello»: «World»
}
```

FastAPI was able to retrieve the header value. Since there was no default value specified (we put in an ellipsis), the header is required. If it's missing, once again, you'll get a 422 status error response.

Additionally, notice that FastAPI automatically converts the header name to *lowercase*. Besides that, since header names are separated by a hyphen, -, most of the time, it also automatically converts it to snake case. Therefore, it works out of the box with any valid Python variable name. The following example shows this behavior by retrieving the User-Agent header:

chapter3_headers_cookies_02.py

```
@app.get("/")
async def get_header(user_agent: str = Header(...)):
    return {"user_agent": user_agent}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_headers_cookies_02.py

Now, let's make a very simple request. We'll keep the default user agent of HTTPie to see what happens:

```
$ http -v GET http://localhost:8000
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/2.4.0

HTTP/1.1 200 OK
content-length: 29
content-type: application/json
date: Mon, 29 Mar 2021 13:37:57 GMT
server: uvicorn

{
  «user_agent»: "HTTPie/2.4.0"
}
```

One very special case of header is cookies. You could retrieve them by parsing the `Cookie` header yourself, but that would be a bit tedious. FastAPI provides another parameter function that automatically does it for you.

The following example simply retrieves a cookie named `hello`:

chapter3_headers_cookies_03.py

```
@app.get("/")
async def get_cookie(hello: Optional[str] = Cookie(None)):
    return {"hello": hello}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_headers_cookies_03.py

Notice that we type hinted the argument as `Optional`, and we set a default value of `None` to the `Cookie` function. This way, even if the cookie is not set in the request, FastAPI will proceed and not generate a 422 status error response.

Headers and cookies can be very useful tools in which to implement some authentication features. In *Chapter 7, Managing Authentication and Security in FastAPI*, you'll learn that there are built-in security functions that can help you to implement common authentication schemes.

The request object

Sometimes, you might find that you need to access a raw request object with all of the data associated with it. That's possible. Simply declare an argument on your path operation function type hinted with the `Request` class:

chapter3_request_object_01.py

```
from fastapi import FastAPI, Request

app = FastAPI()

@app.get("/")
async def get_request_object(request: Request):
    return {"path": request.url.path}
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3\_request\_object\_01.py
```

Under the hood, this is the `Request` object from Starlette, which is a library that provides all the core server logic for FastAPI. You can view a complete description of the methods and properties of this object in the official documentation of Starlette (<https://www.starlette.io/requests/>).

Congratulations! You have now learned all of the basics regarding how to handle request data in FastAPI. As you learned, the logic is the same no matter what part of the HTTP request you want to look at. Simply name the argument you want to retrieve, add a type hint, and use a parameter function to tell FastAPI where it should look. You can even add some validation logic!

In the next section, we'll explore the other side of a REST API job: returning a response.

Customizing the response

In the previous sections, you learned that directly returning a dictionary or a pydantic object in your path operation function was enough for FastAPI to return a JSON response.

Most of the time, you'll want to customize this response a bit further; for instance, by changing the status code, raising validation errors, and setting cookies. FastAPI offers different ways to do this, from the simplest case to the most advanced one. First, we'll learn how to customize the response declaratively by using path operation parameters.

Path operation parameters

In the *Creating the first endpoint and running it locally* section, you learned that in order to create a new endpoint, you had to put a decorator on top of the path operation function. This decorator accepts a lot of options, including ones to customize the response.

The status code

The most obvious thing to customize in an HTTP response is the **status code**. By default, FastAPI will always set a 200 status when everything goes well during your path operation function execution.

Sometimes, it might be useful to change this status. For example, it's good practice in a REST API to return a 201 `Created` status when the execution of the endpoint ends up in the creation of a new object.

To set this, simply specify the `status_code` argument on the path decorator:

chapter3_response_path_parameters_01.py

```
from fastapi import FastAPI, status
from pydantic import BaseModel

class Post(BaseModel):
    title: str

app = FastAPI()

@app.post("/posts", status_code=status.HTTP_201_CREATED)
async def create_post(post: Post):
    return post
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_path_parameters_01.py

The decorator arguments come right after the path as keyword arguments. The `status_code` option simply expects an integer representing the status code. So, we could have written `status_code=201`, but FastAPI provides a useful list in the `status` sub-module that improves code comprehensiveness, as you can see here.

We can try this endpoint to obtain the resulting status code:

```
$ http POST http://localhost:8000/posts title="Hello"
HTTP/1.1 201 Created
content-length: 17
content-type: application/json
date: Tue, 30 Mar 2021 07:56:22 GMT
server: uvicorn

{
  "title": "Hello"
}
```

We have got our 201 status code.

It's important to understand that this option to override the status code is only useful *when everything goes well*. Even if your input data was invalid, you would still get a 422 status error response.

Another interesting scenario for this option is when you have nothing to return, such as when you typically delete an object. In this case, the 204 No content status code is a good fit. In the following example, we implement a simple DELETE endpoint that sets this response status code:

chapter3_response_path_parameters_02.py

```
# Dummy database
posts = {
    1: Post(title="Hello", nb_views=100),
}

@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_
CONTENT)
async def delete_post(id: int):
    posts.pop(id, None)
    return None
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_path_parameters_02.py

Notice that you can very well return `None` in your path operation function. FastAPI will take care of it and return a response with an empty body.

In the *Setting the status code dynamically* section, you'll learn how to customize the status code dynamically inside the path operation logic.

The response model

With FastAPI, the main use case is to directly return a pydantic model that automatically gets turned into properly formatted JSON. However, quite often, you'll find that there are some differences between the input data, the data you store in your database, and the data you want to show to the end user. For instance, perhaps some fields are private or only for internal use, or perhaps some fields are only useful during the creation process and then discarded afterward.

Now, let's consider a simple example. Assume you have a database containing blog posts. Those blog posts have several properties, such as a title, content, or creation date. Additionally, you store the number of views of each one, but you don't want the end user to see any of this.

You could take the standard approach as follows:

chapter3_response_path_parameters_03.py

```
from fastapi import FastAPI
from pydantic import BaseModel

class Post(BaseModel):
    title: str
    nb_views: int

app = FastAPI()

# Dummy database
posts = {
    1: Post(title="Hello", nb_views=100),
}

@app.get("/posts/{id}")
async def get_post(id: int):
    return posts[id]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_path_parameters_03.py

And then call this endpoint:

```
$ http GET http://localhost:8000/posts/1
HTTP/1.1 200 OK
content-length: 32
content-type: application/json
date: Tue, 30 Mar 2021 08:11:11 GMT
server: uvicorn

{
  "nb_views": 100,
  "title": "Hello"
}
```

The `nb_views` property is in the output. However, we don't want this. This is exactly what the `response_model` option is for: to specify another model that only outputs the properties we want. First, let's define another pydantic model with only the `title` property:

`chapter3_response_path_parameters_04.py`

```
class PublicPost(BaseModel):
    title: str
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3\_response\_path\_parameters\_04.py
```

Tip

You might have noticed that we repeat ourselves a lot when defining those models. In *Chapter 4, Managing pydantic Data Models in FastAPI*, you'll learn how to avoid this.

Then, the only change is to add the `response_model` option as a keyword argument for the path decorator:

chapter3_response_path_parameters_04.py

```
@app.get("/posts/{id}", response_model=PublicPost)
async def get_post(id: int):
    return posts[id]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_path_parameters_04.py

Now, let's try to call this endpoint:

```
$ http GET http://localhost:8000/posts/1
HTTP/1.1 200 OK
content-length: 17
content-type: application/json
date: Tue, 30 Mar 2021 08:29:45 GMT
server: uvicorn

{
  "title": "Hello"
}
```

The `nb_views` property is no longer there! Thanks to the `response_model` option, FastAPI automatically converted our `Post` instance into a `PublicPost` instance before serializing it. Now our private data is safe!

The good thing is that this option is also considered by the interactive documentation, which will show the correct output schema to the end user, as you can see in *Figure 3.2*:

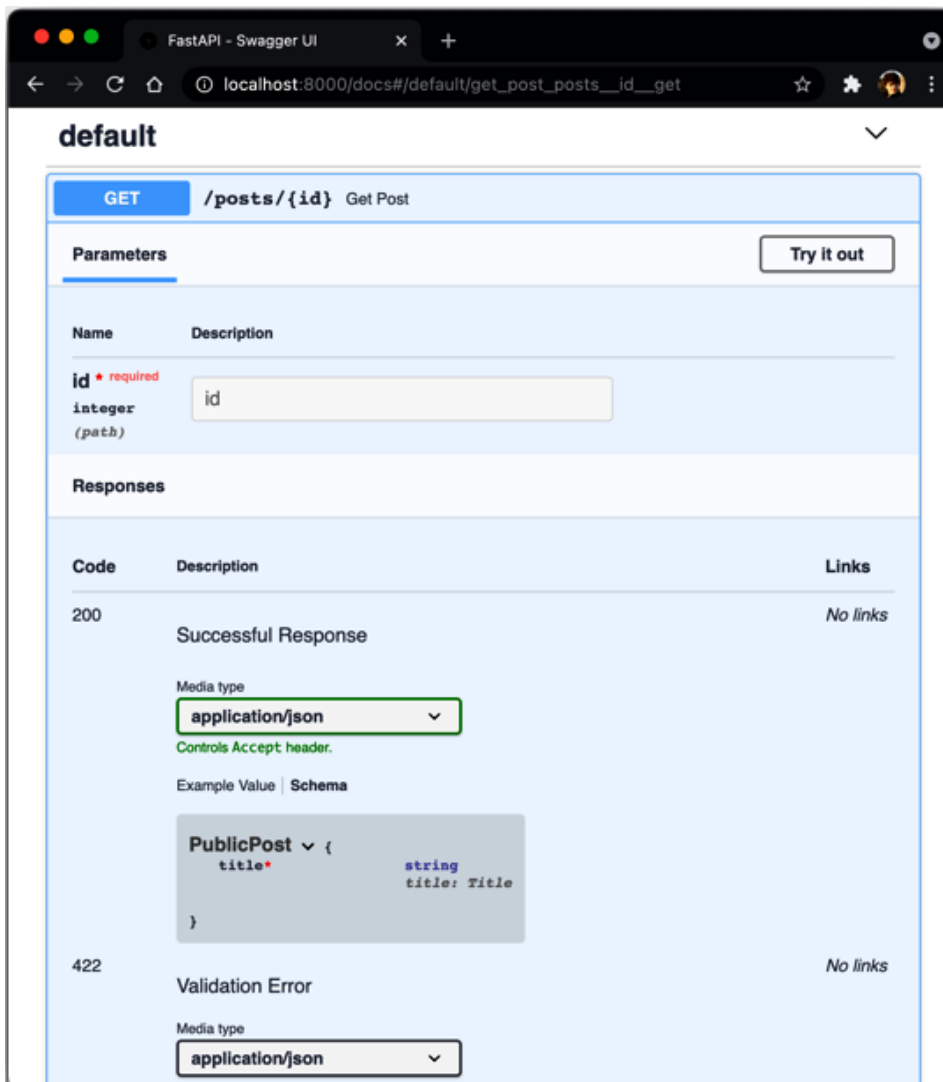


Figure 3.2 – The response model schema in the interactive documentation

So far, you've looked at options that can help you to quickly customize the response generated by FastAPI. Now, we'll introduce another approach that will open up more possibilities.

The response parameter

The body and status code are not the only interesting parts of an HTTP response. Sometimes, it might be useful to return some custom headers or set cookies. This can be done dynamically using FastAPI directly within the path operation logic. How so? By injecting the `Response` object as an argument of the path operation function.

Setting headers

As usual, this only involves setting the proper type hinting to the argument. The following example shows you how to set a custom header:

chapter3_response_parameter_01.py

```
from fastapi import FastAPI, Response

app = FastAPI()

@app.get("/")
async def custom_header(response: Response):
    response.headers["Custom-Header"] = "Custom-Header-Value"
    return {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_parameter_01.py

The `Response` object gives you access to a set of properties, including headers. It's a simple dictionary where the key is the name of the header, and the value is its associated value. Therefore, it's relatively straightforward to set your own custom header.

Also, notice that you *don't have to return* the `Response` object. You can still return JSON-encodable data and FastAPI will take care of forming a proper response, including the headers you've set. Therefore, the `response_model` and `status_code` options we discussed in the *Path operation parameters* section are still honored.

Let's view the result:

```
$ http GET http://localhost:8000
HTTP/1.1 200 OK
content-length: 17
content-type: application/json
custom-header: Custom-Header-Value
date: Wed, 31 Mar 2021 06:22:03 GMT
server: uvicorn

{
  "hello": "world"
}
```

Our custom header is part of the response.

As we mentioned earlier, the good thing about this approach is that it's within your path operation logic. That means you can dynamically set headers depending on what's happening in your business logic.

Setting cookies

Cookies can also be particularly useful when you want to maintain the user's state within the browser between each of their visits.

To prompt the browser to save some cookies in your response, you could, of course, build your own `Set-Cookie` header and set it in the `headers` dictionary, just as we saw in the preceding command block. However, since this can be quite tricky to do, the `Response` object exposes a convenient `set_cookie` method:

`chapter3_response_parameter_02.py`

```
@app.get("/")
async def custom_cookie(response: Response):
    response.set_cookie("cookie-name", "cookie-value", max_age=86400)
    return {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_parameter_02.py

Here, we simply set a cookie, named `cookie-name`, with the value of `cookie-value`. It'll be valid for 86'400 seconds before the browser removes it.

Let's try it:

```
$ http GET http://localhost:8000
HTTP/1.1 200 OK
content-length: 17
content-type: application/json
date: Wed, 31 Mar 2021 06:37:18 GMT
server: uvicorn
set-cookie: cookie-name=cookie-value; Max-Age=86400; Path=/;
SameSite=lax

{
  "hello": "world"
}
```

Here, you can see that we have a nice `Set-Cookie` header with all of the properties of our cookie.

As you may know, cookies have a lot more options than the ones we have shown here; for instance, `path`, `domain`, and `HTTP-only`. The `set_cookie` method supports all of them. You can read about the full list of options in the official Starlette documentation (since `Response` is also borrowed from Starlette) at <https://www.starlette.io/responses/#set-cookie>.

If you're not familiar with the `Set-Cookie` header, we also recommend that you refer to *MDN Web Docs*, which can be accessed at <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>.

Of course, if you need to set several cookies, you can call this method several times.

Setting the status code dynamically

In the *Path operation parameters* section, we discussed a way to declaratively set the status code of the response. The drawback to this approach is that it'll always be the same no matter what's happening inside.

Let's assume that we have an endpoint that updates an object in the database or creates it if it doesn't exist. A good approach would be to return a `200 OK` status when the object already exists or a `201 Created` status when the object has to be created.

To do this, you can simply set the `status_code` property on the `Response` object:

chapter3_response_parameter_03.py

```
# Dummy database
posts = {
    1: Post(title="Hello", nb_views=100),
}

@app.put("/posts/{id}")
async def update_or_create_post(id: int, post: Post, response:
Response):
    if id not in posts:
        response.status_code = status.HTTP_201_CREATED
    posts[id] = post
    return posts[id]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_response_parameter_03.py

First, we check whether the ID in the path exists in the database. If not, we change the status code to 201. Then, we simply assign the post at this ID in the database.

Let's try with an existing post first:

```
$ http PUT http://localhost:8000/posts/1 title="Updated title"
HTTP/1.1 200 OK
content-length: 25
content-type: application/json
date: Wed, 31 Mar 2021 07:02:41 GMT
server: uvicorn

{
  "title": "Updated title"
}
```

The post with an ID of 1 already exists, so we get a 200 status. Now, let's try with a non-existing ID:

```
$ http PUT http://localhost:8000/posts/2 title="Updated title"
HTTP/1.1 201 Created
content-length: 25
content-type: application/json
date: Wed, 31 Mar 2021 07:03:56 GMT
server: uvicorn

{
  "title": "Updated title"
}
```

We get a 201 status!

Now you have a way to dynamically set the status code in your logic. Bear in mind, though, that they *won't be detected by the automatic documentation*. Therefore, they won't appear as a possible response status code in it.

You might be tempted to use this approach to set *error status codes*, such as 400 `Bad Request` or 404 `Not Found`. In fact, you *shouldn't do that*. FastAPI provides a dedicated way to do this: `HTTPException`.

Raising HTTP errors

When calling a REST API, quite frequently, you might find that things don't go very well; you might come across the wrong parameters, invalid payloads, or objects that don't exist anymore. Errors can happen for a lot of reasons.

That's why it's critical to detect them and raise a clear and unambiguous error message to the end user so that they can correct their mistake. In a REST API, there are two very important things that you can use to return an informative message: the status code and the payload.

The status code can give you a precious hint about the nature of the error. Since HTTP protocols provide a wide range of error status codes, your end user might not even need to read the payload to understand what's wrong.

Of course, it's always better to provide a clear error message at the same time in order to give further details and add some useful information regarding how the end user can solve the issue.

Error status codes are crucial

Some APIs choose to always return a 200 status code with the payload containing a property stating whether the request was successful or not, such as `{"success": false}`. Don't do that. The RESTful philosophy encourages you to use the HTTP semantic to give meaning to the data. Having to parse the output and look for a property to determine whether the call was successful is a bad design.

To raise an HTTP error in FastAPI, you'll have to raise a Python exception, `HTTPException`. This exception class will allow us to set a status code and an error message. It is caught by FastAPI error handlers that take care of forming a proper HTTP response.

In the following example, we'll raise a 400 `Bad Request` error if the `password` and `password_confirm` payload properties don't match:

chapter3_raise_errors_01.py

```
@app.post("/password")
async def check_password(password: str = Body(...), password_confirm: str = Body(...)):
    if password != password_confirm:
        raise HTTPException(
            status.HTTP_400_BAD_REQUEST,
            detail="Passwords don't match.",
        )
    return {"message": "Passwords match."}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_raise_errors_01.py

As you can see here, if the passwords are not equal, we directly raise `HTTPException`. The first argument is the status code, and the `detail` keyword argument lets us write an error message.

Let's examine how it works:

```
$ http POST http://localhost:8000/password password="aa"
password_confirm="bb"
HTTP/1.1 400 Bad Request
content-length: 35
content-type: application/json
date: Wed, 31 Mar 2021 11:58:45 GMT
server: uvicorn

{
  "detail": "Passwords don't match."
}
```

Here, we do get a 400 status code and our error message has been wrapped nicely in a JSON object with the `detail` key. This is how FastAPI handles errors by default.

In fact, you are not limited to a simple string for the error message: you can return a dictionary or a list in order to get structured information about the error. For example, take a look at the following code snippet:

chapter3_raise_errors_02.py

```
raise HTTPException(
    status.HTTP_400_BAD_REQUEST,
    detail={
        "message": "Passwords don't match.",
        "hints": [
            "Check the caps lock on your keyboard",
            "Try to make the password visible by clicking on
the eye icon to check your typing",
        ],
    },
)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_raise_errors_02.py

And that's it! You now have the power to raise errors and give meaningful information about them to the end user.

So far, all of the methods you have seen should cover the majority of cases you'll encounter during the development of an API. Sometimes, however, you'll have scenarios where you'll need to build a complete HTTP response yourself. This is the subject of the next section.

Building a custom response

Most of the time, you'll let FastAPI take care of building an HTTP response by simply providing it with some data to serialize. Under the hood, FastAPI uses a subclass of `Response`, called `JSONResponse`. Quite predictably, this response class takes care of serializing some data to JSON and adding the correct `Content-Type` header.

However, there are other response classes that cover common cases:

- `HTMLResponse`: This can be used to return an HTML response.
- `PlainTextResponse`: This can be used to return raw text.
- `RedirectResponse`: This can be used to make a redirection.
- `StreamingResponse`: This can be used to stream a flow of bytes.
- `FileResponse`: This can be used to automatically build a proper file response given the path of a file on the local disk.

You have two ways of using them: either setting the `response_class` argument on the path decorator or directly returning a response instance.

Using the `response_class` argument

This is the simplest and most straightforward way to return a custom response. Indeed, by doing this, you won't even have to create a class instance: you'll just have to return the data as you do usually for standard JSON responses.

This is well suited for `HTMLResponse` and `PlainTextResponse`:

`chapter3_custom_response_01.py`

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

app = FastAPI()
```

```
@app.get("/html", response_class=HTMLResponse)
async def get_html():
    return """
        <html>
            <head>
                <title>Hello world!</title>
            </head>
            <body>
                <h1>Hello world!</h1>
            </body>
        </html>
    """

@app.get("/text", response_class=PlainTextResponse)
async def text():
    return "Hello world!"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_custom_response_01.py

By setting the `response_class` argument on the decorator, you can change the class that will be used by FastAPI to build the response. Then, you can simply return valid data for this kind of response. Notice that the responses classes are imported through the `fastapi.responses` module.

The nice thing about this is that you can combine this option with the ones we saw in the *Path operation parameters* section. Using the `Response` parameter that we described in the *The response parameter* section also works perfectly!

For the other response classes, however, you'll have to build the instance yourself and then return it.

Making a redirection

As mentioned earlier, `RedirectResponse` is a class that helps you build an HTTP redirection, which simply is an HTTP response with a `Location` header pointing to the new URL and a status code in the *3xx range*. It simply expects the URL you wish to redirect to as the first argument:

chapter3_custom_response_02.py

```
@app.get("/redirect")
async def redirect():
    return RedirectResponse("/new-url")
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_custom_response_02.py

By default, it'll use the `307 Temporary Redirect` status code, but you can change this through the `status_code` argument:

chapter3_custom_response_03.py

```
@app.get("/redirect")
async def redirect():
    return RedirectResponse("/new-url", status_code=status.HTTP_301_MOVED_PERMANENTLY)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_custom_response_03.py

Serving a file

Now, let's examine how `FileResponse` works. This will be useful if you wish to propose some files to download. This response class will automatically take care of opening the file on disk and streaming the bytes along with the proper HTTP headers.

Let's take a look at how we can use an endpoint to download a picture of a cat. You'll find this in the code examples repository at <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/assets/cat.jpg>.

For this class to work, first, you'll need another extra dependency, `aiofiles`:

```
$ pip install aiofiles
```

Then, we just need to return an instance of `FileResponse` with the path of the file we want to serve as the first argument:

chapter3_custom_response_04.py

```
@app.get("/cat")
async def get_cat():
    root_directory = path.dirname(path.dirname(__file__))
    picture_path = path.join(root_directory, "assets", "cat.
    jpg")
    return FileResponse(picture_path)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_custom_response_04.py

The `os.path` module

Python provides a module to help you work with file paths, `os.path`. It's the recommended way to manipulate paths, as it takes care of handling them correctly depending on the OS you are running. You can read about the functions of this module in the official documentation at <https://docs.python.org/3/library/os.path.html>.

Let's examine what the HTTP response looks like:

```
$ http GET http://localhost:8000/cat
HTTP/1.1 200 OK
content-length: 71457
content-type: image/jpeg
date: Thu, 01 Apr 2021 06:50:34 GMT
etag: 243d3de0ca74453f0c2d120e2f064e58
last-modified: Mon, 29 Mar 2021 06:40:29 GMT
server: uvicorn
```

```
+-----+
| NOTE: binary data not shown in terminal |
+-----+
```

As you can see, we have the right `Content-Length` and `Content-Type` headers for our image. The response even sets the `Etag` and `Last-Modified` headers so that the browser can properly cache the resource. HTTPie doesn't show the binary data in the body; however, if you open the endpoint in your browser, you'll see the cat appear!

Custom responses

Finally, if you really have a case that's not covered by the provided classes, you always have the option to use the `Response` class to build exactly what you need. With this class, you can set everything, including the body content and the headers.

The following example shows you how to return an XML response:

chapter3_custom_response_05.py

```
@app.get("/xml")
async def get_xml():
    content = """<?xml version="1.0" encoding="UTF-8"?>
    <Hello>World</Hello>
    """
    return Response(content=content, media_type="application/
    xml")
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3/chapter3_custom_response_05.py

You can view the complete list of arguments in the Starlette documentation at <https://www.starlette.io/responses/#response>.

Path operation parameters and response parameters won't have any effect

Bear in mind that when you directly return a `Response` class (or one of its subclasses), the parameters you set on the decorator or the operations you make on the injected `Response` object won't have any effect. They are completely overridden by the `Response` object you return. If you need to customize the status code or the headers, then use the `status_code` and `headers` arguments when instantiating your class.

Well done! Now you have all of the knowledge required to create the response you need for your REST API. You've learned that FastAPI comes with sensible defaults that can help you create proper JSON responses in no time. At the same time, it also gives you access to more advanced objects and options to allow you to make custom responses.

So far, all of the examples we've looked at have been quite short and simple. However, when you're developing a real application, you'll probably have dozens of endpoints and models. In the final section of this chapter, we'll examine how to organize such projects to make them modular and easier to maintain.

Structuring a bigger project with multiple routers

When building a real-world web application, you're likely to have lot of code and logic: data models, API endpoints, and services. Of course, all of those can't live in a single file; we have to structure the project so that it's easy to maintain and evolve.

FastAPI supports the concept of **routers**. They are "sub-parts" of your API and are usually dedicated to a single type of object, such as users or posts, that are defined in their own file. You can then include them in your main FastAPI app so that it can route it accordingly.

In this section, we'll explore how to use routers and how you can structure a FastAPI project. While this structure is one way to do it, and works quite well, it's not a golden rule and can be adapted to your own needs.

In the code examples repository, there is a folder named `chapter3_project`, which contains a sample project with this structure: https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter3_project.

Here is the project structure:

```
.
├── chapter3_project/
│   ├── models/
│   │   ├── __init__.py
│   │   ├── post.py
│   │   └── user.py
│   ├── routers/
│   │   ├── __init__.py
│   │   └── posts.py
```

```
|   └─ users.py
|   └─ __init__.py
|   └─ app.py
|   └─ db.py
```

Here, you can see that we chose to have packages that contain pydantic models on one side and routers on the other side. At the root of the project, we have a file named `app.py` that will expose the main FastAPI application. The `db.py` file defines a dummy database for the sake of the example.

The `__init__.py` files are there to properly define our directories as Python packages. You can read more details about this in the *Packages, modules, and imports* section of *Chapter 2, Python Programming Specificities*.

First, let's examine what a FastAPI router looks like:

users.py

```
from typing import List

from fastapi import APIRouter, HTTPException, status

from chapter3_project.models.user import User, UserCreate
from chapter3_project.db import db

router = APIRouter()

@router.get("/")
async def all() -> List[User]:
    return list(db.users.values())
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3_project/routers/users.py

As you can see here, instead of instantiating the `FastAPI` class, you instantiate the `APIRouter` class. Then, you can use it exactly the same way to decorate your path operation functions.

Also, notice that we import the pydantic models from the relevant module in the `models` package.

We won't go into detail about the logic of the endpoints, but we invite you to read about it. It uses all the FastAPI features that we've explored so far.

Now, let's take a look at how to import this router and include it within a FastAPI application:

app.py

```
from fastapi import FastAPI

from chapter3_project.routers.posts import router as posts_
router

from chapter3_project.routers.users import router as users_
router

app = FastAPI()

app.include_router(posts_router, prefix="/posts",
tags=["posts"])

app.include_router(users_router, prefix="/users",
tags=["users"])
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter3_project/app.py

As usual, we instantiate the `FastAPI` class. Then, we use the `include_router` method to add our sub-router. You can see that we simply imported the router from its relevant module and used it as the first argument of `include_router`. Notice that we used the syntax as while importing. Since both `users` and `posts` routers are named the same inside their module, this syntax allows us to alias their name and, thus, avoid **name collision**.

Additionally, you can see that we set the keyword argument as `prefix`. This allows us to prefix the path of all the endpoints of this router. This way, you don't have to hardcode it in the router logic and can easily change it for the whole router. It can also be used to provide versioned paths of your API, such as `/v1`.

Finally, the `tags` argument helps you to group endpoints in the interactive documentation for better readability. By doing this, the `posts` and `users` endpoints will be clearly separated in the documentation.

And that's all you need to do! You can run this whole application, as usual, with `uvicorn`:

```
$ uvicorn chapter3_project.app:app
```

If you open the interactive documentation at `http://localhost:8000/docs`, you'll see that all the routes are there, grouped by the tags we specified when including the router:

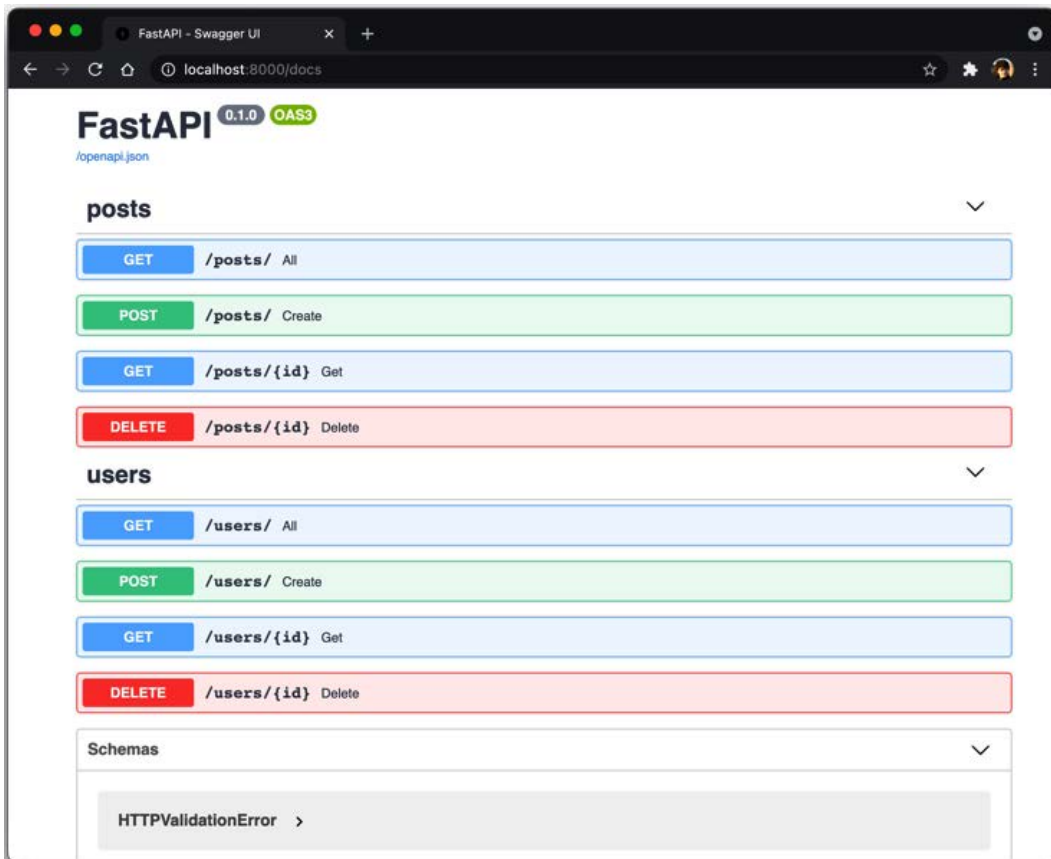


Figure 3.3 – Tagged routers in the interactive documentation

Once again, you can see that FastAPI is both powerful and very lightweight to use. The good thing about routers is that you can even nest them, that include sub-routers in routers that include other routers themselves. Therefore, you can have a quite complex routing hierarchy with very low effort.

Summary

Well done! You're now acquainted with all the basic features of FastAPI. Throughout this chapter, you've learned how to create and run API endpoints where you can validate and retrieve data from all parts of an HTTP request: the path, the query, the parameters, the headers, and, of course, the body. You've also learned how to tailor the HTTP response to your needs, whether it is a simple JSON response, an error, or a file to download. Finally, you looked at how to define separate API routers and include them in your main application to keep a clean and maintainable project structure.

You have enough knowledge now to start building your own API with FastAPI. In the next chapter, we'll focus on pydantic models. You now know that they are at the core of the data validation features of FastAPI, so it's crucial to fully understand how they work and how to manipulate them efficiently.

4

Managing Pydantic Data Models in FastAPI

This chapter will cover in more detail the definition of a data model with Pydantic, the underlying data validation library used by FastAPI. We'll explain how to implement variations of the same model without repeating the same code again and again, thanks to class inheritance. Finally, we'll show how to implement custom data validation logic into Pydantic models.

In this chapter, we're going to cover the following main topics:

- Defining models and their field types with Pydantic
- Creating model variations with class inheritance
- Adding custom data validation with Pydantic
- Working with Pydantic objects

Technical requirements

To run the code examples, you'll need a Python virtual environment, which we set up in *Chapter 1, Python Development Environment Setup*.

You'll find all the code examples of this chapter in the dedicated GitHub repository at <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter4>.

Defining models and their field types with Pydantic

Pydantic is a powerful library for defining data models using Python classes and type hints. This approach makes those classes completely compatible with static type checking. Besides, since there are regular Python classes, we can use inheritance and also define our very own methods to add custom logic.

In *Chapter 3, Developing a RESTful API with FastAPI*, you learned the basics of defining a data model with Pydantic: you have to define a class inheriting from `BaseModel` and list all the fields as class properties, each one with a proper type hint to enforce their type.

In this section, we'll focus on model definition and see all the possibilities we have to define the fields.

Standard field types

We'll begin by defining fields with standard types, which only involve simple type hints. Let's review a simple model representing information about a person. You can see this in the following code sample:

chapter4_standard_field_types_01.py

```
from pydantic import BaseModel

class Person(BaseModel):
    first_name: str
    last_name: str
    age: int
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_01.py

As we said, you just have to write the name of the fields and type-hint it with the intended type. Of course, we are not limited to scalar types: we can actually use compound types such as lists, tuples, or datetime classes. In the following example, you can see a model using those more complex types:

chapter4_standard_field_types_02.py

```
from datetime import date
from enum import Enum
from typing import List

from pydantic import BaseModel, ValidationError

class Gender(Enum):
    MALE = "MALE"
    FEMALE = "FEMALE"
    NON_BINARY = "NON_BINARY"

class Person(BaseModel):
    first_name: str
    last_name: str
    gender: Gender
    birthdate: date
    interests: List[str]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_02.py

There are three things to notice in this example.

First, we used the standard Python Enum class as a type for the gender field. This allows us to specify a set of valid values. If we input a value that's not in this enumeration, Pydantic will raise an error, as illustrated in the following example:

chapter4_standard_field_types_02.py

```
# Invalid gender
try:
```

```

Person(
    first_name="John",
    last_name="Doe",
    gender="INVALID_VALUE",
    birthdate="1991-01-01",
    interests=["travel", "sports"],
)
except ValidationError as e:
    print(str(e))

```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_02.py

If you run the preceding example, you'll get this output:

```

1 validation error for Person
gender
  value is not a valid enumeration member; permitted:
  'MALE', 'FEMALE', 'NON_BINARY' (type=type_error.enum; enum_
  values=[<Gender.MALE: 'MALE'>, <Gender.FEMALE: 'FEMALE'>,
  <Gender.NON_BINARY: 'NON_BINARY'>])

```

Actually, this is exactly what we already did in *Chapter 3, Developing a RESTful API with FastAPI*, to limit the allowed values of the path parameter.

Then, we used the date Python class as a type for the `birthdate` field. Pydantic is able to automatically parse dates and datetimes given as an **International Organization for Standardization (ISO)** format string or a timestamp integer and instantiate a proper date or datetime object. Of course, if the parsing fails, you'll also get an error. You can experiment with this in the following example:

chapter4_standard_field_types_02.py

```

# Invalid birthdate
try:
    Person(
        first_name="John",
        last_name="Doe",
        gender=Gender.MALE,

```

```
        birthdate="1991-13-42",
        interests=["travel", "sports"],
    )
except ValidationError as e:
    print(str(e))
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_02.py

And here is the output:

```
1 validation error for Person
birthdate
  invalid date format (type=value_error.date)
```

Finally, we defined `interests` as a list of strings. Once again, Pydantic will check if the field is a valid list of strings.

Obviously, if everything is okay, we get a `Person` instance and have access to the properly parsed fields. This is what we show in the following code sample:

chapter4_standard_field_types_02.py

```
# Valid
person = Person(
    first_name="John",
    last_name="Doe",
    gender=Gender.MALE,
    birthdate="1991-01-01",
    interests=["travel", "sports"],
)
# first_name='John' last_name='Doe' gender=<Gender.MALE: 'MALE'> birthdate=datetime.date(1991, 1, 1)
interests=['travel', 'sports']
print(person)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_02.py

As you see, this is quite powerful, and we can have quite complex field types. But that's not all: fields can be Pydantic models themselves, allowing you to have sub-objects! In the following code example, we expand the previous one to add an address field:

chapter4_standard_field_types_03.py

```
class Address(BaseModel):
    street_address: str
    postal_code: str
    city: str
    country: str

class Person(BaseModel):
    first_name: str
    last_name: str
    gender: Gender
    birthdate: date
    interests: List[str]
    address: Address
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_03.py

We just have to define another Pydantic model and use it as a type hint. Now, you can either instantiate a `Person` instance with an already valid `Address` instance or, even better, with a dictionary. In this case, Pydantic will automatically parse it and validate it against the address model.

In the following code sample, we try to input an invalid address:

chapter4_standard_field_types_03.py

```
try:
    Person(
        first_name="John",
        last_name="Doe",
        gender="INVALID_VALUE",
        birthdate="1991-01-01",
```

```

        interests=["travel", "sports"],
        address={
            "street_address": "12 Squirell Street",
            "postal_code": "424242",
            "city": "Woodtown",
            # Missing country
        }
    )
except ValidationError as e:
    print(str(e))

```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_03.py

This will generate the following validation error:

```

1 validation error for Person
  address -> country
    field required (type=value_error.missing)

```

Pydantic clearly shows the missing field in the sub-object. Once again, if everything goes well, we get a Person instance and its associated Address, as you can see in the following extract:

chapter4_standard_field_types_03.py

```

# Valid
person = Person(
    first_name="John",
    last_name="Doe",
    gender=Gender.MALE,
    birthdate="1991-01-01",
    interests=["travel", "sports"],
    address={
        "street_address": "12 Squirell Street",
        "postal_code": "424242",
        "city": "Woodtown",

```

```
        "country": "US",
    },
)
print(person)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_standard_field_types_03.py

Optional fields and default values

Up to now, we've assumed that each field had to be provided when instantiating the model. Quite often, however, there are values that we want to be optional because they may not be relevant for each object instance. Sometimes, we also wish to set a default value for a field when it's not specified.

As you may have guessed, this is done quite simply, with the `Optional` typing annotation, as illustrated in the following code sample:

`chapter4_optional_fields_default_values_01.py`

```
from typing import Optional

from pydantic import BaseModel

class UserProfile(BaseModel):
    nickname: str
    location: Optional[str] = None
    subscribed_newsletter: bool = True

user = UserProfile(nickname="jdoe")
print(user) # nickname='jdoe' location=None subscribed_
newsletter=True
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_optional_fields_default_values_01.py

When defining a field with the `Optional` type hint, it accepts a `None` value. As you see in the preceding code sample, the default value can be simply assigned by putting the value after an equals sign.

Be careful, though: don't assign default values such as this for dynamic types such as datetimes. By doing so, the datetime instantiation will be evaluated only once when the model is imported. The effect of this is that all the objects you'll instantiate will then share the same value instead of having a fresh value. You can observe this behavior in the following example:

chapter4_optional_fields_default_values_02.py

```
class Model(BaseModel):
    # Don't do this.
    # This example shows you why it doesn't work.
    d: datetime = datetime.now()

o1 = Model()
print(o1.d)

time.sleep(1) # Wait for a second

o2 = Model()
print(o2.d)

print(o1.d < o2.d) # False
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_optional_fields_default_values_02.py

Even though we waited for 1 second between the instantiation of `o1` and `o2`, the `d` datetime is the same! This means that the datetime is evaluated once when the class is imported.

You can have the same kind of problem if you want to have a default list, such as `l: List[str] = ["a", "b", "c"]`. Notice that this is true for every Python object, not only Pydantic models, so you should bear this in mind.

So, how to assign dynamic default values? Fortunately, Pydantic provides a `Field` function that allows us to set some advanced options on our fields, including one to set a factory for creating dynamic values. Before showing you this, we'll first introduce the `Field` function.

Field validation

In *Chapter 3, Developing a RESTful API with FastAPI*, we showed how to apply some validation to the `request` parameters to check if a number was in a certain range or if a string was matching a **regular expression (regex)**. Actually, those options directly come from Pydantic! We can use the same ones to apply validation to the fields of a model.

To do this, we'll use the `Field` function from Pydantic and use its result as the default value of the field. In the following example, we define a `Person` model with the `first_name` and `last_name` required properties, which should be at least three characters long, and an optional `age` property, which should be an integer between 0 and 120. We show the implementation of this model in the following code sample:

chapter4_fields_validation_01.py

```
from typing import Optional

from pydantic import BaseModel, Field, ValidationError

class Person(BaseModel):
    first_name: str = Field(..., min_length=3)
    last_name: str = Field(..., min_length=3)
    age: Optional[int] = Field(None, ge=0, le=120)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_fields_validation_01.py

As you see, the syntax is very similar to the one we saw for `Path`, `Query`, and `Body`. The first positional argument defines the *default value* for the field. If the field is required, we use the ellipsis `...`. Then, the keyword arguments are there to set options for the field, including some basic validation.

You can view a complete list of the arguments accepted by `Field` in the official Pydantic documentation, at <https://pydantic-docs.helpmanual.io/usage/schema/#field-customisation>.

Dynamic default values

In the previous section, we warned you about setting dynamic values as defaults. Fortunately, Pydantic provides the `default_factory` argument on the `Field` function to cover this use case. This argument expects you to pass a function that will be called during model instantiation. Thus, the resulting object will be evaluated at runtime each time you create a new object. You can see how to use it in the following example:

chapter4_fields_validation_02.py

```
from datetime import datetime
from typing import List

from pydantic import BaseModel, Field

def list_factory():
    return ["a", "b", "c"]

class Model(BaseModel):
    l: List[str] = Field(default_factory=list_factory)
    d: datetime = Field(default_factory=datetime.now)
    l2: List[str] = Field(default_factory=list)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_fields_validation_02.py

You simply have to pass a function to this argument. Don't put arguments on it—it'll be Pydantic that will automatically call the function for you when instantiating a new object. If you need to call a function with specific arguments, you'll have to wrap it into your own function, as we did for `list_factory`.

Notice also that the first positional argument used for the default value (such as `None` or `...`) is completely omitted here. This makes sense: it's not consistent to have both a default value and a factory. Pydantic will raise an error if you set those two arguments together.

Validating email addresses and URLs with Pydantic types

For convenience, Pydantic provides some classes to use as field types to validate some common patterns such as email addresses or **Uniform Resource Locators (URLs)**.

In the following example, we'll use `EmailStr` and `HttpUrl` to validate an email address and a **HyperText Transfer Protocol (HTTP)** URL.

For `EmailStr` to work, you'll need an optional dependency, `email-validator`, which you can install with the following command:

```
$ pip install email-validator
```

Those classes work like any other type or class: just use them as a type hint for your field. You can see this in the following extract:

chapter4_pydantic_types_01.py

```
from pydantic import BaseModel, EmailStr, HttpUrl,
ValidationError
```

```
class User(BaseModel):
    email: EmailStr
    website: HttpUrl
```

```
https://github.com/PacktPublishing/Building-Data-Science-
Applications-with-FastAPI/blob/main/chapter4/chapter4_
pydantic_types_01.py
```

In the following example, we check that the email address is correctly validated:

chapter4_pydantic_types_01.py

```
# Invalid email
try:
    User(email="jdoe", website="https://www.example.com")
except ValidationError as e:
    print(str(e))
```

```
https://github.com/PacktPublishing/Building-Data-Science-
Applications-with-FastAPI/blob/main/chapter4/chapter4_
pydantic_types_01.py
```

You will see the following output:

```
1 validation error for User
email
  value is not a valid email address (type=value_error.email)
```

We also check that the URL is correctly parsed, as follows:

chapter4_pydantic_types_01.py

```
# Invalid URL
try:
    User(email="jdoe@example.com", website="jdoe")
except ValidationError as e:
    print(str(e))
```

```
https://github.com/PacktPublishing/Building-Data-Science-
Applications-with-FastAPI/blob/main/chapter4/chapter4_
pydantic_types_01.py
```

You will see the following output:

```
1 validation error for User
website
  invalid or missing URL scheme (type=value_error.url.scheme)
```

If you have a look at a valid example, shown next, you'll see that the URL is parsed into an object, giving you access to the different parts of it, such as the scheme or hostname:

chapter4_pydantic_types_01.py

```
# Valid
user = User(email=»jdoe@example.com«, website=»https://www.
example.com«)
# email='jdoe@example.com' website=HttpUrl('https://www.
example.com', scheme='https', host='www.example.com',
tld='com', host_type='domain')
print(user)
```

```
https://github.com/PacktPublishing/Building-Data-Science-
Applications-with-FastAPI/blob/main/chapter4/chapter4_
pydantic_types_01.py
```

Pydantic provides a quite big set of types that can help you in various situations. We invite you to review a full list of these in the official documentation, at <https://pydantic-docs.helpmanual.io/usage/types/#pydantic-types>.

You now have a better view of how to define finely your Pydantic models, by using more advanced types or leveraging the validation features. As we said, those models are at the heart of FastAPI, and you'll probably have to define several variations for the same entity to account for several situations. In the next section, we'll show how to do that with minimum repetition.

Creating model variations with class inheritance

In *Chapter 3, Developing a RESTful API with FastAPI*, we saw a case where we needed to define two variations of a Pydantic model in order to split between the data we want to store in the backend and the data we want to show to the user. This is a common pattern in FastAPI: you define one model for creation, one for the response and one for the data to store in the database.

We show this basic approach in the following sample:

chapter4_model_inheritance_01.py

```
from pydantic import BaseModel

class PostCreate(BaseModel):
    title: str
    content: str

class PostPublic(BaseModel):
    id: int
    title: str
    content: str

class PostDB(BaseModel):
    id: int
    title: str
    content: str
    nb_views: int = 0
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_model_inheritance_01.py

We have three models here, covering three situations. These are outlined as follows:

- `PostCreate` will be used for a `POST` endpoint to create a new post. We expect the user to give the title and the content; however, the **identifier (ID)** will be automatically determined by the database.
- `PostPublic` will be used when we retrieve the data of a post. We want its title and content, of course, but also its associated ID in the database.
- `PostDB` will carry all the data we wish to store in the database. Here, we also want to store the number of views, but we want to keep this secret to make our own statistics internally.

You can see here that we are repeating ourselves quite a lot, especially with the `title` and `content` fields. In bigger examples with lots of fields and lots of validation options, this could quickly become unmanageable.

The solution here is to leverage model inheritance to avoid this. The approach is simple: identify the fields that are common to every variation and put them in a model that will be used as a base for every other. Then, you only have to inherit from that model to create your variations and add the specific fields. In the following example, we see what our previous example looks like with this method:

`chapter4_model_inheritance_02.py`

```
from pydantic import BaseModel

class PostBase(BaseModel):
    title: str
    content: str

class PostCreate(PostBase):
    pass
```

```
class PostPublic(PostBase):  
    id: int  
  
class PostDB(PostBase):  
    id: int  
    nb_views: int = 0
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_model_inheritance_02.py

Now, whenever you need to add a field for the whole entity, all you have to do is to add it to the `PostBase` model.

It's also very convenient if you wish to define methods on your model. Remember that Pydantic models are regular Python classes, so you can implement as many methods as you wish!

chapter4_model_inheritance_03.py

```
class PostBase(BaseModel):  
    title: str  
    content: str  
  
    def excerpt(self) -> str:  
        return f"{self.content[:140]}..."
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_model_inheritance_03.py

Defining the `excerpt` method on `PostBase` means that this will be available in every model variation.

While not strictly required, this inheritance approach is strongly recommended to avoid code duplication and, ultimately, bugs. We'll see in the next section that it'll make even more sense with custom validation methods.

Adding custom data validation with Pydantic

Up to now, we've seen how to apply basic validation to our models, through the `Field` arguments or the custom types provided by Pydantic. In a real-world project, though, you'll probably need to add your own custom validation logic for your specific case. Pydantic allows this by defining **validators**, which are methods on the model that can be applied at a field level or an object level.

Applying validation at a field level

This is the most common case: have a validation rule for a single field. To define it in Pydantic, we'll just have to write a static method on our model and decorate it with the `validator` decorator. As a reminder, decorators are syntactic sugar, allowing the wrapping of a function or a class with common logic, without compromising readability.

The following example checks a birth date by verifying that the person is not more than 120 years old:

chapter4_custom_validation_01.py

```
from datetime import date

from pydantic import BaseModel, validator

class Person(BaseModel):
    first_name: str
    last_name: str
    birthdate: date

    @validator("birthdate")
    def valid_birthdate(cls, v: date):
        delta = date.today() - v
        age = delta.days / 365
        if age > 120:
            raise ValueError("You seem a bit too old!")
        return v
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_custom_validation_01.py

As you see here, the `validator` is a static class method (the first argument, `cls`, being the class itself), with the value to validate as the `v` argument. It's decorated by the `validator` decorator, which expects the name of the argument to validate as the first argument.

Pydantic expects two things for this method, detailed as follows:

- If the value is not valid according to your logic, you should raise a `ValueError` error with an explicit error message.
- Otherwise, you should return the value that will be assigned in the model. Notice that it doesn't need to be the same as the input value: you can very well change it to fit your needs. That's actually what we'll do in an upcoming section, *Applying validation before Pydantic parsing*.

Applying validation at an object level

It happens quite often that the validation of one field is dependent on another—for example, to check if a password confirmation matches the password or to enforce a field to be required in certain circumstances. To allow this kind of validation, we need to access the whole object data. For this, Pydantic provides the `root_validator` decorator, which is illustrated in the following code example:

chapter4_custom_validation_02.py

```
from pydantic import BaseModel, EmailStr, ValidationError,
root_validator

class UserRegistration(BaseModel):
    email: EmailStr
    password: str
    password_confirmation: str

    @root_validator()
    def passwords_match(cls, values):
        password = values.get("password")
        password_confirmation = values.get("password_
confirmation")
        if password != password_confirmation:
```

```
raise ValueError("Passwords don't match")  
return values
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_custom_validation_02.py

The usage of this decorator is similar to the `validator` decorator. The static `class` method is called along with the `values` argument, which is a *dictionary* containing all the fields. Thus, you can retrieve each one of them and implement your logic.

Once again, Pydantic expects two things for this method, outlined as follows:

- If the values are not valid according to your logic, you should raise a `ValueError` error with an explicit error message.
- Otherwise, you should return a `values` dictionary that will be assigned to the model. Notice that you could change some values in this dictionary to fit your needs.

Applying validation before Pydantic parsing

By default, your validators are run after Pydantic has done its parsing work. This means that the value you get already conforms to the type of field you specified. If the type is incorrect, Pydantic raises an error without calling your validator.

However, you may sometimes wish to provide some custom parsing logic that allows you to transform input values that would have been incorrect for the type you set. In that case, you would need to run your validator before the Pydantic parser: this is the purpose of the `pre` argument on `validator`.

In the following example, we show how to transform a string with values separated by a comma into a proper list:

chapter4_custom_validation_03.py

```
from typing import List  
  
from pydantic import BaseModel, validator  
  
class Model(BaseModel):  
    values: List[int]
```

```
@validator("values", pre=True)
def split_string_values(cls, v):
    if isinstance(v, str):
        return v.split(",")
    return v

m = Model(values="1,2,3")
print(m.values) # [1, 2, 3]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_custom_validation_03.py

You see here that our validator first checks whether we have a string. If we do, we split a comma-separated string and return the resulting list; otherwise, we directly return the value. Pydantic will run its parsing logic after, so you can still be sure that an error will be raised if `v` is an invalid value.

Working with Pydantic objects

When developing API endpoints with FastAPI, you'll likely get a lot of Pydantic model instances to handle. It's then up to you to implement the logic to make a link between those objects and your services, such as your database or your **machine learning (ML)** model. Fortunately, Pydantic provides methods to make this very easy. We'll review common use cases that will be useful for you during development.

Converting an object into a dictionary

This is probably the action you'll perform the most on a Pydantic object: convert it to a raw dictionary that'll be easy to send to another API or use in a database, for example. You just have to call the `dict` method on the object instance.

The following example reuses the `Person` and `Address` models we saw in the *Standard field types* section of this chapter:

chapter4_working_pydantic_objects_01.py

```
person = Person(
    first_name="John",
    last_name="Doe",
    gender=Gender.MALE,
```

```
birthdate="1991-01-01",
interests=["travel", "sports"],
address={
    "street_address": "12 Squirell Street",
    "postal_code": "424242",
    "city": "Woodtown",
    "country": "US",
},
)

person_dict = person.dict()
print(person_dict["first_name"]) # "John"
print(person_dict["address"]["street_address"]) # "12 Squirell
Street"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_01.py

As you see, calling `dict` is enough to transform the whole data into a dictionary. Sub-objects are also recursively converted: the `address` key points itself to a dictionary with the address properties.

Interestingly, the `dict` method supports some arguments, allowing you to select a subset of properties to be converted. You can either state the ones you want to be included or the ones you want to exclude, as you can see in the following sample:

chapter4_working_pydantic_objects_02.py

```
person_include = person.dict(include={"first_name", "last_
name"})
print(person_include) # {"first_name": "John", "last_name":
"Doe"}

person_exclude = person.dict(exclude={"birthdate",
"interests"})
print(person_exclude)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_02.py

The `include` and `exclude` arguments expect a set with the keys of the fields you want to include or exclude.

For nested structures such as `address` here, you can also use a dictionary to specify which sub-field you want to include or exclude, as illustrated in the following example:

chapter4_working_pydantic_objects_02.py

```
person_nested_include = person.dict(
    include={
        "first_name": ...,
        "last_name": ...,
        "address": {"city", "country"},
    }
)
# {"first_name": "John", "last_name": "Doe", "address":
# {"city": "Woodtown", "country": "US"}}
print(person_nested_include)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_02.py

The resulting `address` dictionary only contains the city and the country. Notice that when using this syntax, scalar fields such as `first_name` or `last_name` have to be associated with the ellipsis `...`

If you use a conversion quite often, it can be interesting to put it in a method so that you can reuse it at will, as illustrated in the following example:

chapter4_working_pydantic_objects_03.py

```
class Person(BaseModel):
    first_name: str
    last_name: str
    gender: Gender
    birthdate: date
    interests: List[str]
    address: Address
```

```
def name_dict(self):  
    return self.dict(include={"first_name", "last_name"})
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_03.py

Creating an instance from a sub-class object

In the earlier section, *Creating model variations with class inheritance*, we studied the common pattern of having specific model classes depending on the situation. In particular, you'll have a model dedicated for the creation endpoint, with only the required fields for creation, and a database model with all the fields we want to store.

Let's take again the Post example, as follows:

chapter4_working_pydantic_objects_04.py

```
class PostBase(BaseModel):  
    title: str  
    content: str  
  
class PostCreate(PostBase):  
    pass  
  
class PostPublic(PostBase):  
    id: int  
  
class PostDB(PostBase):  
    id: int  
    nb_views: int = 0
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_04.py

In our path operation function for our create endpoint, we'll thus get a PostCreate instance with only title and content. However, we need to build a proper PostDB instance before storing it in the database.

A convenient way to do this is to jointly use the `dict` method and the unpacking syntax. In the following example, we implemented a creation endpoint using this approach:

chapter4_working_pydantic_objects_04.py

```
@app.post("/posts", status_code=status.HTTP_201_CREATED,
response_model=PostPublic)
async def create(post_create: PostCreate):
    new_id = max(db.posts.keys() or (0,)) + 1

    post = PostDB(id=new_id, **post_create.dict())

    db.posts[new_id] = post
    return post
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_04.py

As you see, the path operation function would give us a valid `PostCreate` object. Then, we want to transform it into a `PostDB` object.

We first determine the missing `id` property, which is given to us by the database. Here, we use a dummy database based on a dictionary, so we simply take the maximum key already present in the database and increment it. In a real-world situation, this would have been automatically determined by the database.

The most interesting line here is the `PostDB` instantiation. You see that we first assign the missing fields by the `keyword` argument and then unpack the dictionary representation of `post_create`. As a reminder, the effect of `**` in a function call is to transform a dictionary such as `{"title": "Foo", "content": "Bar"}` into keyword arguments such as this: `title="Foo", content="Bar"`. It's a very convenient and dynamic approach to set all the fields we already have into our new model.

Notice also that we set the `response_model` argument on the path operation decorator. We already explained this in *Chapter 3, Developing a RESTful API with FastAPI*, but basically, it prompts FastAPI to build a **JSON** response with only the fields of `PostPublic`, even though we return a `PostDB` instance at the end of the function.

Updating an instance with a partial one

In some situations, you'll want to allow partial updates. In other words, you'll allow the end user to only send the fields they want to change to your API and omit the ones that shouldn't change. This is the usual way of implementing a PATCH endpoint.

To do this, you would first need a special Pydantic model with all the fields marked as optional so that no error is raised when a field is missing. Let's see what this looks like with our `Post` example, as follows:

chapter4_working_pydantic_objects_05.py

```
class PostBase(BaseModel):
    title: str
    content: str

class PostPartialUpdate(BaseModel):
    title: Optional[str] = None
    content: Optional[str] = None
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_05.py

We are now able to implement an endpoint that will accept a subset of our `Post` fields. Since it's an update, we'll retrieve an existing post in the database thanks to its ID. Then, we'll have to find a way to only update the fields in the payload and keep the others untouched. Fortunately, Pydantic once again has this covered, with handy methods and options.

Let's see how the implementation of such an endpoint could look in the following example:

chapter4_working_pydantic_objects_05.py

```
@app.patch("/posts/{id}", response_model=PostPublic)
async def partial_update(id: int, post_update:
    PostPartialUpdate):
    try:
        post_db = db.posts[id]

        updated_fields = post_update.dict(exclude_unset=True)
```

```
        updated_post = post_db.copy(update=updated_fields)
        db.posts[id] = updated_post
        return updated_post
    except KeyError:
        raise HTTPException(status.HTTP_404_NOT_FOUND)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter4/chapter4_working_pydantic_objects_05.py

Our path operation function takes two arguments: the `id` property (from the path), and a `PostPartialUpdate` instance (from the body).

The first thing to do is to check if this `id` property exists in the database. Since we use a dictionary for our dummy database, accessing a non-existing key will raise a `KeyError` error. If this happens, we simply raise an `HTTPException` exception with the 404 status code.

Now for the interesting part: updating the existing object. You see that the first thing we do is transform `PostPartialUpdate` into a dictionary with the `dict` method. This time, however, we set the `exclude_unset` argument to `True`. The effect of this is that Pydantic won't output the fields that were not provided in the resulting dictionary: we only get the fields that the user did send in the payload.

Then, on our existing `post_db` database instance, we call the `copy` method. This is a useful method to clone a Pydantic object into another instance. The nice thing about this method is that it even accepts an `update` argument. This argument expects a dictionary with all the fields that should be updated during the copy: that's exactly what we want to do with our `updated_fields` dictionary!

And that's it! We now have an `updated post` instance with only the changes required in the payload. You'll probably use the `exclude_unset` argument and the `copy` method quite often while developing with FastAPI, so be sure to keep them in mind—they'll make your life easier!

Summary

Congratulations! You've learned another important aspect of FastAPI: designing and managing data models with Pydantic. You should now be confident about creating models and applying validation at a field level, with built-in options and types, and also by implementing your own validation methods. You also know how to apply validation at an object level to check consistency between several fields. You also reviewed a common pattern, leveraging model inheritance to avoid code duplication and repetition while defining your model variations. Finally, you learned how to correctly work with Pydantic model instances in order to transform and update them in an efficient and readable way.

You know almost all the features of FastAPI by now. There is a last very powerful one for you to learn: **dependency injections**. These will allow you to define your own logic and values to directly inject into your path operation functions, as you do for path parameters and payload objects, which you'll be able to reuse everywhere in your project. That's the subject of the next chapter.

5

Dependency Injections in FastAPI

In this chapter, we'll focus on one of the most interesting parts of FastAPI: **dependency injections**. You'll see that it is a powerful and readable approach to reuse logic across your project. Indeed, it will allow you to create complex building blocks for your project that you'll be able to use everywhere in your logic. An authentication system, a query parameters' validator, or a rate-limiter are typical use cases for dependencies. In FastAPI, a dependency injection can even call another one recursively, allowing you to build high-level blocks from basic features. By the end of this chapter, you'll be able to create your own dependencies for FastAPI and use them at several levels of your project.

In this chapter, we're going to cover the following main topics:

- What is dependency injection?
- Creating and using a function dependency
- Creating and using a parameterized dependency with a class
- Using dependencies at a path, router, and global level

Technical requirements

You'll need a Python virtual environment, as we set up in *Chapter 1, Python Development Environment Setup*.

You'll find all the code examples of this chapter in the dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter5>.

What is dependency injection?

Generally speaking, **dependency injection** is a system able to automatically instantiate objects and the ones they depend on. The responsibility of developers is then to only provide a declaration of how an object should be created, and let the system resolve all the dependency chains and create the actual objects at runtime.

FastAPI allows you to declare the objects and variables you wish to have at hand only by declaring them in the path operation function arguments. Actually, we already used dependency injection in the previous chapters. In the following example, we use the `Header` function to retrieve the `user-agent` header:

chapter5_what_is_dependency_injection_01.py

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/")
async def header(user_agent: str = Header(...)):
    return {"user_agent": user_agent}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_what_is_dependency_injection_01.py

Internally, the `Header` function has some logic to automatically get the request object, check for the required header, return its value, or raise an error if it's not present. From the developer's perspective, however, we don't know how it handled the required objects for this operation: we just ask for the value we need. *That's dependency injection.*

Admittedly, you could reproduce this example quite easily in the function body by picking the `user-agent` property in the headers dictionary of the `Request` object. However, the dependency injection approach has numerous advantages over this:

- The *intent is clear*: you know what the endpoint expects in the request data without reading the function's code.
- You have a *clear separation of concern between the logic of the endpoint and the more generic logic*: the header retrieval and the associated error handling doesn't pollute the rest of the logic; it's self-contained in the dependency function. Besides, it can be reused easily in other endpoints.
- In the case of FastAPI, it's used to *generate the OpenAPI schema* so that the automatic documentation can clearly show which parameters are expected for this endpoint.

Put another way, whenever you need utility logic to retrieve or validate data, make security checks or call external logic that you'll need several times across your application, a dependency is an ideal choice.

FastAPI relies heavily on this dependency injection system and encourages developers to use it to implement their building blocks. It may be a bit puzzling if you come from other web frameworks such as Flask or Express, but you'll surely be quickly convinced by its power and relevance.

To convince you, we'll now see how you can create and use your very own dependency, in the form of a function to begin with.

Creating and using a function dependency

In FastAPI, a dependency can be defined either as a function or as a callable class. In this section, we'll focus on the functions, which are the ones you'll probably work with most of the time.

As we said, a dependency is a way to wrap some logic that will retrieve some sub-values or sub-objects, make something with them, and finally return a value that will be injected into the endpoint calling it.

Let's look at a first example where we define a function dependency to retrieve the pagination query parameters, `skip` and `limit`:

chapter5_function_dependency_01.py

```
async def pagination(skip: int = 0, limit: int = 10) ->
    Tuple[int, int]:
    return (skip, limit)

@app.get("/items")
async def list_items(p: Tuple[int, int] = Depends(pagination)):
    skip, limit = p
    return {"skip": skip, "limit": limit}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_function_dependency_01.py

There are two parts of this example:

- First, we have the dependency definition, with the `pagination` function. You see that we define two arguments, `skip` and `limit`, which are integers with default values. Those will be the query parameters on our endpoint. We define them exactly like we would have done on a path operation function. That's the beauty of this approach: FastAPI will recursively handle the arguments on the dependency and match them with the request data, such as query parameters or headers, if needed.

We simply return those values as a tuple.

- Secondly, we have the path operation function, `list_items`, that uses the `pagination` dependency. You see here that the usage is quite similar to what we have done for header or body values: we define the name of our resulting argument and we use a function result as a default value. In the case of a dependency, we use the `Depends` function. Its role is to take a function in the argument and execute it when the endpoint is called. The sub-dependencies are automatically discovered and executed.

In the endpoint, we have the `pagination` directly in the form of a tuple.

Let's run this example with the following command:

```
$ uvicorn chapter5.chapter5_function_dependency_01:app
```

Now, we'll try to call the `/items` endpoint and see whether it's able to retrieve the query parameters. You can try this with the following HTTPie command:

```
$ http "http://localhost:8000/items?limit=5&skip=10"
HTTP/1.1 200 OK
content-length: 21
content-type: application/json
date: Sat, 29 May 2021 16:03:36 GMT
server: uvicorn

{
  "limit": 5,
  "skip": 10
}
```

The `limit` and `skip` query parameters have correctly been retrieved thanks to our function dependency. You can also try to call the endpoint without the query parameter and notice that it will return you the default values.

Type hint of a dependency return value

You may have noticed that we had to type hint the result of our dependency in the path operation arguments, even though we already type hinted the dependency function itself. Unfortunately, this is a limitation of FastAPI and its `Depends` function, which isn't able to forward the type of the dependency function. Therefore, we have to type hint the result by hand, as we did here.

And that's it! As you see, it's very simple and straightforward to create and use a dependency in FastAPI. Of course, you can now reuse it at will in several endpoints, as you can see in the rest of our examples:

chapter5_function_dependency_01.py

```
@app.get("/items")
async def list_items(p: Tuple[int, int] = Depends(pagination)):
    skip, limit = p
    return {"skip": skip, "limit": limit}

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

```
async def list_things(p: Tuple[int, int] =
    Depends(pagination)):
    skip, limit = p
    return {"skip": skip, "limit": limit}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_function_dependency_01.py

Of course, we can do more complex things in those dependencies, just like we would in a regular path operation function. In the following example, we add some validation to those pagination parameters and cap the limit at 100:

chapter5_function_dependency_02.py

```
async def pagination(
    skip: int = Query(0, ge=0),
    limit: int = Query(10, ge=0),
) -> Tuple[int, int]:
    capped_limit = min(100, limit)
    return (skip, capped_limit)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_function_dependency_02.py

As you can see, our dependency starts to become more complex:

- We added the `Query` function to our arguments to add a validation constraint: now, an error 422 will be raised if `skip` or `limit` are negative integers.
- We ensure that the limit is, at most, 100.

The code on our path operation functions doesn't have to change: we have a clear separation of concern between the logic of the endpoint and the more generic logic for the pagination parameters.

Let's see another typical use of dependencies: *get an object or raise a 404 error*.

Get an object or raise a 404 error

In a REST API, you'll typically have endpoints to get, update, and delete a single object given its identifier in the path. On each one, you'll likely have the same logic: try to retrieve this object in the database or raise an error 404 if it doesn't exist. That's a perfect use case for a dependency! In the following example, you'll see how to implement it:

chapter5_function_dependency_03.py

```
async def get_post_or_404(id: int) -> Post:
    try:
        return db.posts[id]
    except KeyError:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_function_dependency_03.py

The dependency definition is simple: it takes in an argument the ID of the post we want to retrieve. It will be pulled from the corresponding path parameter. Then, we check whether it exists in our dummy dictionary database: if it does, we return it, otherwise, we raise an `HTTPException` with the status code 404.

That's the key takeaway of this example: you can raise errors in your dependencies. It's extremely useful to check for some pre-conditions before your endpoint logic is executed. Another typical example for this is authentication: if the endpoint requires a user to be authenticated, we can raise a 401 error in the dependency by checking for the token or the cookie.

Now, we can use this dependency in each of our API endpoints, as you can see in the following example:

chapter5_function_dependency_03.py

```
@app.get("/posts/{id}")
async def get(post: Post = Depends(get_post_or_404)):
    return post

@app.patch("/posts/{id}")
```

```
async def update(post_update: PostUpdate, post: Post =
    Depends(get_post_or_404)):
    updated_post = post.copy(update=post_update.dict())
    db.posts[post.id] = updated_post
    return updated_post

@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_
    CONTENT)
async def delete(post: Post = Depends(get_post_or_404)):
    db.posts.pop(post.id)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_function_dependency_03.py

As you can see, we just had to define the `post` argument and use the `Depends` function on our `get_post_or_404` dependency. Then, within the path operation logic, we are guaranteed to have our `post` object at hand and we can focus on our core logic, which is now very concise. The `get` endpoint, for example, just has to return the object.

In this case, the only point of attention is to not forget the ID parameter in the path of those endpoints. According to the rules of FastAPI, if you don't set this parameter in the path, it will automatically be regarded as a query parameter, which is not what we want here. You can find more details about this in the *Path parameters* section of *Chapter 3, Developing a RESTful API with FastAPI*.

That's all for the function dependencies. As we said, those are the main building blocks in a FastAPI project. In some cases, however, you'll need to have some parameters on those dependencies, for example, with values coming from environment variables. For this, we can define class dependencies.

Creating and using a parameterized dependency with a class

In the previous section, we defined dependencies as regular functions, which works well in most cases. Still, you may need to set some parameters on a dependency to finely tune its behavior. Since the arguments of the function are set by the dependency injection system, we can't add an argument to the function.

In the pagination example, we added some logic to cap the limit value at 100. If we wanted to set this maximum limit dynamically, how would we do that?

The solution is to create a class that will be used as a dependency. This way, we can set class properties, with the `__init__` method, for example, and use them in the logic of the dependency itself. This logic will be defined in the `__call__` method of the class. If you remember what we learned in the *Callable object* section of *Chapter 2, Python Programming Specificities*, you know that it makes the object callable, meaning it can be called like a regular function. Actually, that is all that `Depends` requires for a dependency: being a callable. We'll use this property to create a parameterized dependency thanks to a class.

In the following example, we reimplemented the pagination example with a class, allowing us to set the maximum limit dynamically:

chapter5_class_dependency_01.py

```
class Pagination:
    def __init__(self, maximum_limit: int = 100):
        self.maximum_limit = maximum_limit

    async def __call__(
        self,
        skip: int = Query(0, ge=0),
        limit: int = Query(10, ge=0),
    ) -> Tuple[int, int]:
        capped_limit = min(self.maximum_limit, limit)
        return (skip, capped_limit)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_class_dependency_01.py

As you can see, the logic in the `__call__` method is the same as in the function we defined in the previous example. The only difference here is that we can pull our maximum limit from our class properties that we can set at the object initialization.

Then, you can simply create an instance of this class and use it as a dependency with `Depends` on your path operation function, as you can see in the following code block:

chapter5_class_dependency_01.py

```
pagination = Pagination(maximum_limit=50)

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



```
async def list_items(p: Tuple[int, int] = Depends(pagination)):  
    skip, limit = p  
    return {"skip": skip, "limit": limit}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_class_dependency_01.py

Here, we hardcoded the value 50, but we could very well pull it from a configuration file or an environment variable.

The other advantage of a class dependency is that it can maintain local values in memory. This property can be very useful if we have to make some heavy initialization logic, such as loading a machine learning model, for example, that we want to do only once at startup. Then, the callable part just has to call the loaded model to make the prediction, which should be quite fast.

Use class methods as dependencies

Even if the `__call__` method is the most straightforward way to make a class dependency, you can directly pass a method to `Depends`. Indeed, as we said, it simply expects a callable as an argument, and a class method is a perfectly valid callable!

This approach can be very useful if you have common parameters or logic that you need to reuse in slightly different cases. For example, you could have one pre-trained machine learning model made with Scikit-learn. Before applying the decision function, you may want to apply different pre-process steps depending on the input data.

To do this, simply write your logic in a class method and pass it to the `Depends` function through the dot notation.

You can see this in the following example, where we implement another style for our pagination dependency, with `page` and `size` parameters instead of `skip` and `limit`:

chapter5_class_dependency_02.py

```
class Pagination:  
    def __init__(self, maximum_limit: int = 100):  
        self.maximum_limit = maximum_limit  
  
    async def skip_limit(  
        self,  
        skip: int = Query(0, ge=0),
```

```
        limit: int = Query(10, ge=0),
    ) -> Tuple[int, int]:
        capped_limit = min(self.maximum_limit, limit)
        return (skip, capped_limit)

    async def page_size(
        self,
        page: int = Query(1, ge=1),
        size: int = Query(10, ge=0),
    ) -> Tuple[int, int]:
        capped_size = min(self.maximum_limit, size)
        return (page, capped_size)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_class_dependency_02.py

The logic of the two methods is quite similar. We just look at different query parameters. Then, on our path operation functions, we set the `/items` endpoint to work with the `skip/limit` style, while the `/things` endpoint will work with the `page/size` style:

chapter5_class_dependency_02.py

```
pagination = Pagination(maximum_limit=50)

@app.get («/items»)
async def list_items(p: Tuple[int, int] = Depends(pagination.
skip_limit)):
    skip, limit = p
    return {"skip": skip, "limit": limit}

@app.get ("/things")
async def list_things(p: Tuple[int, int] = Depends(pagination.
page_size)):
    page, size = p
    return {"page": page, "size": size}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_class_dependency_02.py

As you see, we only have to pass the method we wish through the dot notation on the `pagination` object.

To sum up, the class dependency approach is more advanced than the function approach but can be very useful for cases when you need to set parameters dynamically, perform heavy initialization logic, or reuse common logic on several dependencies.

Until now, we've assumed that we care about the return value of the dependency. While this will probably be the case most of the time, you may occasionally need to call a dependency to check for some conditions, but don't really need the returned value. FastAPI allows such use cases, and that's what we'll see now.

Using dependencies at a path, router, and global level

As we said, dependencies are the recommended way to create building blocks in a FastAPI project, allowing you to reuse logic across endpoints while maintaining maximum code readability. Until now, we've applied them on a single endpoint, but couldn't we expand this approach to a whole router? Or even a whole FastAPI application? Actually, we can!

The main motivation for this is to be able to apply some global request validation or perform side logic on several routes without the need to add the dependency on each endpoint. Typically, an authentication method or a rate-limiter could be very good candidates for this use case.

To show you how it works, we'll implement a simple dependency that we will use across all the following examples. You can see it in the following example:

chapter5_path_dependency_01.py

```
def secret_header(secret_header: Optional[str] = Header(None))
    -> None:
    if not secret_header or secret_header != "SECRET_VALUE":
        raise HTTPException(status.HTTP_403_FORBIDDEN)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_path_dependency_01.py

This dependency will simply look for a header in the request named `Secret-Header`. If it's missing or not equal to `SECRET_VALUE`, it will raise a 403 error. Please note that this approach is only for the sake of the example; there are better ways to secure your API, which we'll cover in *Chapter 7, Managing Authentication and Security in FastAPI*.

Use a dependency on a path decorator

Until now, we've assumed that we were always interested in the return value of the dependency. As our `secret_header` dependency clearly shows here, this is not always the case. This is why you can add a dependency on a path operation decorator instead of the arguments. You can see how in the following example:

`chapter5_path_dependency_01.py`

```
@app.get("/protected-route", dependencies=[Depends(secret_header)])
async def protected_route():
    return {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_path_dependency_01.py

The path operation decorator accepts an argument, `dependencies`, which expects a list of dependencies. You see that, just like for dependencies you pass in arguments, you need to wrap your function (or callable) with the `Depends` function.

Now, whenever the `/protected-route` route is called, the dependency will be called and will check for the required header.

As you may have guessed, since `dependencies` is a list, you can add as many dependencies as you need.

That's interesting, but what if we want to protect a whole set of endpoints? It would be a bit cumbersome and error-prone to add it manually on each one. Fortunately, FastAPI provides a way to do that.

Use a dependency on a whole router

If you recall the *Structure a bigger project with multiple routers* section in *Chapter 3, Developing a RESTful API with FastAPI*, you know that you can create several routers in your project to clearly split the different parts of your API and "wire" them to your main FastAPI application. This is done with the `APIRouter` class and the `include_router` method of the `FastAPI` class.

With this approach, it can be interesting to inject a dependency on the whole router, so that it's called for every route of this router. You have two ways of doing this:

- Set the `dependencies` argument on the `APIRouter` class, as you can see in the following example:

`chapter5_router_dependency_01.py`

```
router = APIRouter(dependencies=[Depends(secret_header)])

@router.get("/route1")
async def router_route1():
    return {"route": "route1"}

@router.get("/route2")
async def router_route2():
    return {"route": "route2"}

app = FastAPI()
app.include_router(router, prefix="/router")
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_router_dependency_01.py

- Set the `dependencies` argument on the `include_router` method, as you can see in the following example:

chapter5_router_dependency_02.py

```
router = APIRouter()

@router.get("/route1")
async def router_route1():
    return {"route": "route1"}

@router.get("/route2")
async def router_route2():
    return {"route": "route2"}

app = FastAPI()
app.include_router(router, prefix="/router",
dependencies=[Depends(secret_header)])
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_router_dependency_02.py

In both cases, the `dependencies` argument expects a list of dependencies. You see that, just like for dependencies you pass in arguments, you need to wrap your function (or callable) with the `Depends` function. Of course, since it's a list, you can add several dependencies if you need.

Now, how to choose between the two approaches? In both cases, the effect will be exactly the same, so we could say it doesn't really matter. Philosophically, we could say that we should declare a dependency on the `APIRouter` class if it's needed in the context of this router. Put another way, we could ask ourselves the question, *Does this router work without this dependency if we run it independently?* If the answer to this question is *no*, then you should probably set the dependency on the `APIRouter` class. Otherwise, declaring it in the `include_router` method may make more sense. But again, this is an intellectual choice that won't change the functionality of your API, so feel free to choose the one you're more comfortable with.

We are now able to set dependencies for a whole router. In some cases, it could also be interesting to declare them for a whole application!

Use a dependency on a whole application

If you have a dependency that implements some logging or rate-limiting functionality, for example, it could be interesting to execute it for every endpoint of your API. Fortunately, FastAPI allows this, as you can see in the following example:

chapter5_global_dependency_01.py

```
app = FastAPI(dependencies=[Depends(secret_header)])
```

```
@app.get("/route1")
```

```
async def route1():
```

```
    return {"route": "route1"}
```

```
@app.get("/route2")
```

```
async def route2():
```

```
    return {"route": "route2"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter5/chapter5_global_dependency_01.py

Once again, you only have to set the `dependencies` argument directly on the main `FastAPI` class. Now, the dependency is applied to every endpoint in your API!

In *Figure 5.1*, we propose a simple decision tree to determine at which level you should inject your dependency:

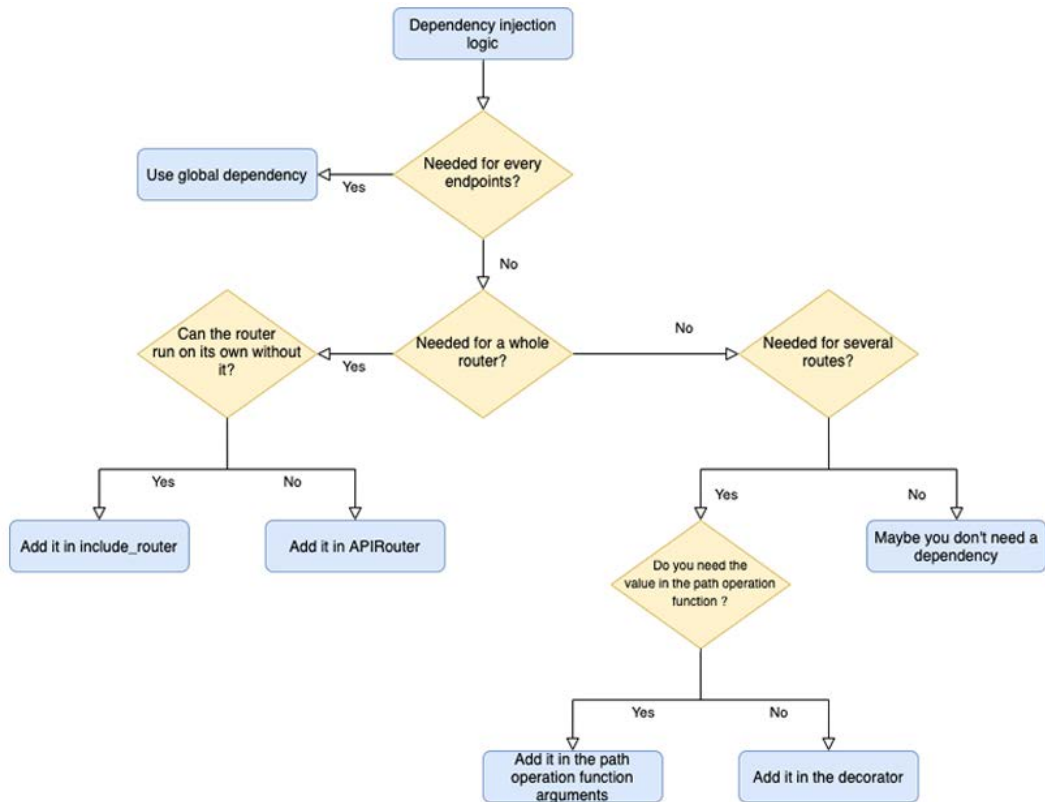


Figure 5.1 – At which level should I inject my dependency?

Summary

Well done! You should now be comfortable with one of the most iconic features of FastAPI: dependency injections. By implementing your own dependencies, you'll be able to keep common logic that you wish to reuse across your API separate from the endpoints' logic. This will make your project clean and maintainable while retaining maximum readability: dependencies just need to be declared as arguments of the path operation functions, which will help to understand the intent without having to read the body of the function.

Those dependencies can be both simple wrappers to retrieve and validate request parameters, or complex services performing machine learning tasks. Thanks to the class-based approach, you can indeed set dynamic parameters or keep a local state for your most advanced tasks.

Finally, those dependencies can also be used at a router or global level, allowing you to perform common logic or checks for a set of routes or a whole application.

That's the end of the first part of this book! You're now acquainted with the main features of FastAPI and should now be able to write clean and performant REST APIs with the framework.

In the next part, we'll take your knowledge to the next level and show you how you can implement and deploy a robust, secure, and tested web backend. The first chapter will be dedicated to databases, a must-have for most APIs to be able to read and write data.

Section 2: Build and Deploy a Complete Web Backend with FastAPI

The goal of this section is to show you how to build a real-world backend with FastAPI that can read and write data and authenticate users, and that is properly tested and correctly configured for a production environment.

This section comprises the following chapters:

- *Chapter 6, Databases and Asynchronous ORMs*
- *Chapter 7, Managing Authentication and Security in FastAPI*
- *Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI*
- *Chapter 9, Testing an API Asynchronously with pytest and HTTPX*
- *Chapter 10, Deploying a FastAPI Project*

6

Databases and Asynchronous ORMs

The main goal of a REST API is, of course, to read and write data. So far, we've solely worked with the tools given by Python and FastAPI, allowing us to build reliable endpoints to process and answer requests. However, we haven't been able to effectively retrieve and persist that information: we missed a **database**.

The goal of this chapter is to show you how you can interact with different types of databases and related libraries inside FastAPI. It's worth noting that FastAPI is completely agnostic regarding databases: you can use any system you want and it's your responsibility to integrate it. This is why we'll review three different approaches to integrate a database, that is, using basic SQL queries, using **Object-Relational Mapping (ORM)**, and, finally, using a NoSQL database.

In this chapter, we're going to cover the following main topics:

- An overview of relational and NoSQL databases
- Communicating with a SQL database with SQLAlchemy
- Communicating with a SQL database with Tortoise ORM
- Communicating with a MongoDB database using Motor

Technical requirements

For this chapter, you'll require a Python virtual environment, just as we set up in *Chapter 1, Python Development Environment Setup*.

For the *Communicating with a MongoDB database using Motor* section, you'll need a running MongoDB server on your local computer. The easiest way to do this is to run it as a Docker container. If you've never used Docker before, we recommend that you refer to the *Getting started* tutorial in the official documentation at <https://docs.docker.com/get-started/>. Once you have done this, you'll be able to run a MongoDB server using this simple command:

```
$ docker run -d --name fastapi-mongo -p 27017:27017 mongo:4.4
```

The MongoDB server instance will then be available on your local computer at port 27017.

You can find all the code examples for this chapter in the dedicated GitHub repository at <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter6>.

An overview of relational and NoSQL databases

The role of a database is to store data in a structured way, preserve the integrity of the data, and offer a query language that enables you to retrieve this data when an application needs it.

Nowadays, when it comes to choosing a database for your web project, you have two main choices: **relational databases**, with their associated SQL query language, and **NoSQL databases**, named in opposition to the first category.

Selecting the right technology for your project is left up to you, as it greatly depends on your needs and requirements. In this section, we'll outline the main characteristics and features of those two database families and try to give you some insights into choosing the right one for your project.

Relational databases

Relational databases have existed since the 1970s, and they have proved to be very performant and reliable over time. They are almost inseparable from the SQL query language, which has become the de facto standard for querying such databases. Even if there are a few differences between one database engine and another, most of the syntax is common, simple to understand, and flexible enough to express complex queries.

Relational databases implement the relational model: each entity, or object, of the application is stored in **tables**. For example, if we consider a blog application, we could have tables that represent *users*, *posts*, and *comments*.

Each of those tables has several **columns** representing the attributes of the entity. If we consider the posts, we could have a *title*, a *publication date*, and *content*. In those tables, there will be several rows, each one representing a single entity of this type; each post will have its own row.

One of the key points of relational databases is, as their name suggests, *relationships*. Each table can be in relation to others, with rows referring to other rows in other tables. In our example, a post could be related to the user who wrote it. In the same way, a comment could be linked to the post that it relates to.

The main motivation behind this is to *avoid duplication*. Indeed, it wouldn't be very efficient to repeat the user's name or email on each of its posts. If it needs to be modified at some point, we would have to go through each post, which is error-prone and puts data consistency at risk. This is why we prefer to *reference* the user in the posts. So, how can we do this?

Usually, each row in a relational database has an identifier, called a **primary key**. This is unique in the table and will allow you to uniquely identify this row. Therefore, it's possible to use this key in another table to reference it. We call it a **foreign key**: the key is foreign in the sense that it refers to another table.

In *Figure 6.1*, you can view a representation of such database schema using an entity-relationship diagram. Note that each table has its own primary key or `id`. The `Post` table refers to a `User`, through the `user_id` foreign key. Similarly, the `Comment` table refers to both a `post` and a `user` through the `user_id` and `post_id` foreign keys:

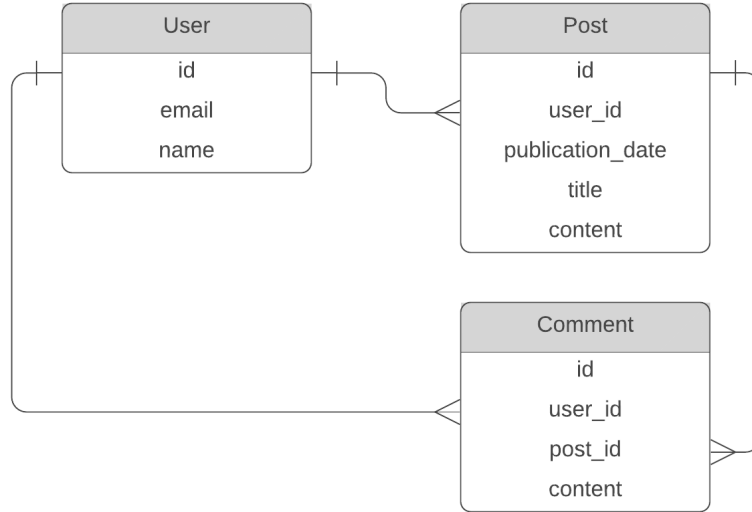


Figure 6.1 – A relational database schema example for a blog application

In an application, you'll likely want to retrieve a post with the comments and the user associated. To do so, we perform a **join query**, which will return all the relevant records based on the foreign keys. Relational databases are designed to perform such tasks efficiently; however, those operations can become expensive if the schema is more complex. This is why it's important to carefully design a relational schema and its queries.

NoSQL databases

All database engines that are not relational fall back into the NoSQL category. In fact, this is a quite vague denomination that regroups different families of databases: key-value stores, such as Redis; graph databases, such as Neo4j; and document-oriented databases, such as MongoDB. That said, most of the time when we talk about "NoSQL databases", we are implicitly referring to document-oriented databases. They are the ones that interest us in this chapter.

Document-oriented databases move away from the relational architecture and try to store all the information of a given object inside a single **document**. As such, performing a join query is much rarer and usually more difficult.

Those documents are stored in **collections**. Contrary to relational databases, documents in a collection might not have all of the same attributes: while tables in relational databases have a defined schema, collections accept any kind of document.

In *Figure 6.2*, you can view a representation of our previous blog example, which has been adapted into a document-oriented database structure. In this configuration, we have chosen to have a collection for users and another one for posts. However, notice that the comments are now part of a post, that is, they are included as a list:

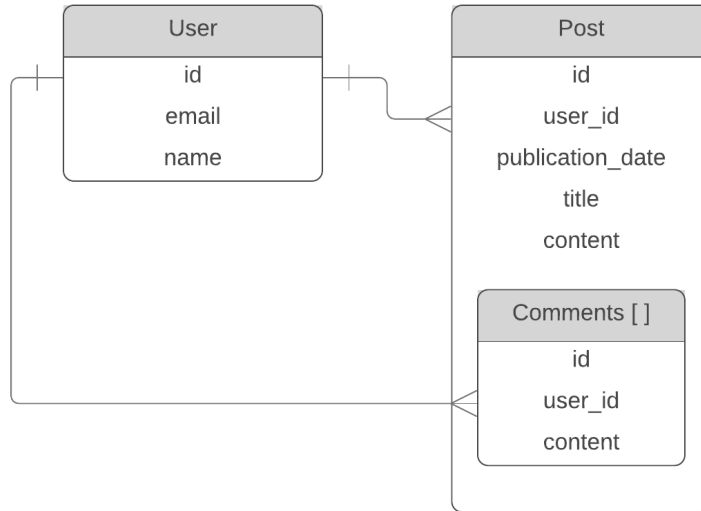


Figure 6.2 — A document-oriented schema example for a blog application

To retrieve a post and all of its comments, you don't need to perform a join query: all the data comes in one query. This was the main motivation behind the development of document-oriented databases: increase the query performance by limiting the need to look at several collections. In particular, this has been adapted for applications with huge data scales and less structured data, such as social networks.

Which one should you choose?

As we mentioned in the introduction to this section, the choice of database engine greatly depends on your application and needs. A detailed comparison between relational and document-oriented databases is beyond the scope of this book, but here are some elements for you to think about.

Relational databases are very good for storing structured data with a lot of relationships between the entities. Besides, they maintain data consistency at all costs, even in the event of errors or hardware failures. However, you'll have to precisely define your schema and consider a migration system to update your schema if your needs evolve.

On the other hand, document-oriented databases don't require you to define a schema: they accept any document structure, so it can be convenient if your data is highly variable or if your project is not mature enough. The downside of this is that they are far less picky in terms of data consistency, which could result in data loss or inconsistencies.

For small and medium-sized applications, the choice doesn't really matter: both relational databases and document-oriented databases are very optimized and will deliver awesome performance at such scales.

Now, we'll show you how to work with those different kinds of databases using FastAPI. When we introduced asynchronous I/O in *Chapter 2, Python Programming Specificities*, we mentioned that it was important to carefully select the libraries you use to perform I/O operations. Of course, databases are particularly important in this context!

While working with classic non-async libraries is perfectly possible in FastAPI, you could miss out one of the key aspects of the framework and might not reach the best performance it can offer. That's why, in this chapter, we'll only focus on async libraries.

Communicating with a SQL database with SQLAlchemy

To begin, we'll discuss how to work with a relational database using the SQLAlchemy library. SQLAlchemy has been around for years and is the most popular library in Python when you wish to work with SQL databases.

In this chapter, it's worth noting that *we'll only consider the core part of the library*, which only provides the tools to abstract communication with a SQL database. We won't consider the ORM part, as, in the next section, we'll focus on another ORM: Tortoise. As such, in this section, we'll pay very close attention to the SQL language.

Recently, async support has been added in version 1.4 but is not yet considered stable. That's why, for now, we'll combine it with the *databases* library by Encode, the same team behind Starlette, which provides an asynchronous connection layer for SQLAlchemy. In *Figure 6.3*, we have presented a schema for you to better visualize the interaction between the different libraries:

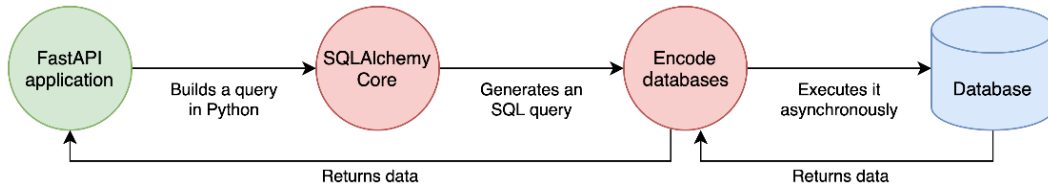


Figure 6.3 – The interaction between SQLAlchemy Core and the Encode databases

The first step is to install this library:

```
$ pip install databases[sqlite]
```

This will install the `databases` library, SQLAlchemy, and the required drivers to work with SQLite databases. SQLite is a very convenient relational engine that stores all of the data inside a single file on your computer, which is perfect for testing and experimenting. Unlike PostgreSQL or MySQL, you don't need to install and run a complex server.

Each type of SQL server will require its own driver, which provides specific instructions on which to communicate with them. Of course, the ones for PostgreSQL and MySQL are provided by `databases`, which will be useful when building a real-world project. You can check the installation instructions in the official documentation at <https://www.encode.io/databases/>.

Now, we'll show you, step by step, how to set up a complete database interaction. *Figure 6.4* shows you the structure of the project:

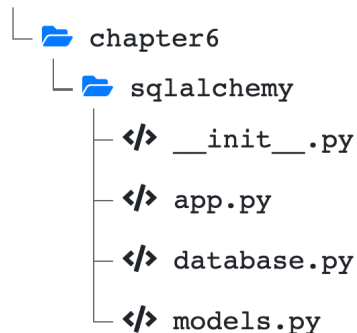


Figure 6.4 – The FastAPI and SQLAlchemy project structure

Creating the table schema

First, you need to define the SQL schema for your tables: the name, the columns, and their associated types and properties. SQLAlchemy provides a full set of classes and functions to help you in this task. In the following example, you can view the definition of the `posts` table:

models.py

```
metadata = sqlalchemy.MetaData()

posts = sqlalchemy.Table(
    "posts",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_
key=True, autoincrement=True),
    sqlalchemy.Column("publication_date", sqlalchemy.
DateTime(), nullable=False),
    sqlalchemy.Column("title", sqlalchemy.String(length=255),
nullable=False),
    sqlalchemy.Column("content", sqlalchemy.Text(),
nullable=False),
)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/models.py>

First, let's create a `metadata` object. Its role is to keep all the information of a database schema together. This is why you should create it only once in your whole project and always use the same one throughout.

Next, we will define a table using the `Table` class. The first argument is the name of the table, followed by the `metadata` object. Then, we list all of the columns that should be defined in our table, thanks to the `Column` class. The first argument is the name of the column, followed by its type and a certain number of options. For example, we define our `id` column as a primary key with auto-increment, which is quite common in a SQL database.

Note that we won't go through all the types and options provided by SQLAlchemy. Just know that they closely follow the ones that are usually provided by SQL databases. You can check the complete list in the official documentation, as follows:

- You can find the list of types at https://docs.sqlalchemy.org/en/13/core/type_basics.html#generic-types
- You can find the list of Column arguments at https://docs.sqlalchemy.org/en/13/core/metadata.html#:~:text=sqlalchemy.schema.Column.__init__

If you take a look at the code above the table definition, you'll see that we also defined the corresponding Pydantic models for our post entity. Since they will be used by FastAPI to validate the request payload, they must match the SQL definition to avoid any errors from the database when we try to insert a new row later.

Connecting to a database

Now that our table is ready, we have to set up the connection between our FastAPI app and the database engine. To begin, we'll instantiate several objects, as shown in the following example:

database.py

```
DATABASE_URL = "sqlite:///chapter6_sqlalchemy.db"
database = Database(DATABASE_URL)
sqlalchemy_engine = sqlalchemy.create_engine(DATABASE_URL)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/database.py>

Here, you can see that we have set our connection string inside the `DATABASE_URL` variable. Generally, it consists of the database engine, followed by authentication information and the hostname of the database server. You can find an overview of this format in the official SQLAlchemy documentation at <https://docs.sqlalchemy.org/en/13/core/engines.html#database-urls>. In the case of SQLite, we simply have to give the path of the file that will store all of the data.

Then, we instantiate a `Database` instance using this URL. This is the connection layer provided by databases that will allow us to perform asynchronous queries.

We also define `sqlalchemy_engine`, which is the standard synchronous connection object provided by SQLAlchemy. You might think that it constitutes an overlap with `database`, and you would be absolutely right. We'll clarify why we need it in our example later.

Then, we define a simple function whose role is to simply return the `database` instance. This is shown in the following example:

database.py

```
def get_database() -> Database:
    return database
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/database.py>

We'll use this function as a dependency to easily retrieve this instance in our path operation functions.

Using a dependency to retrieve a database instance

You might be wondering why we don't just import the database instance into our app and use it directly rather than passing it through a dependency. In fact, it would totally work. However, it would make our life very hard when trying to implement unit tests. Indeed, it would be very difficult to replace this instance with a mock or test database. With a dependency, FastAPI makes it very easy to swap it with another function. We'll view this in more detail in *Chapter 9, Testing an API Asynchronously with pytest and HTTPX*.

Now, we need to tell FastAPI to open the connection with the database when it starts the application and then close it when exiting. Fortunately, FastAPI provides two special decorators to perform tasks at startup and shutdown, as you can see in the following example:

app.py

```
@app.on_event("startup")
async def startup():
    await database.connect()
    metadata.create_all(sqlalchemy_engine)
```

```
@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

Decorating functions with the `on_event` decorators allows us to trigger some useful logic when FastAPI starts or stops. In this case, we simply call the `connect` and `disconnect` methods of the database accordingly. This will ensure that the database connection is open and ready to process requests.

Additionally, you can see that we call the `create_all` method on the metadata object. This is the same metadata object we defined in the previous section and that we have imported here. The goal of this method is to create the table's schema inside our database. If we don't do that, our database would be empty and we wouldn't be able to save or retrieve data. This method is designed to work with a standard SQLAlchemy engine; this is why we instantiated it earlier. It has no other use in the application.

However, we only created a schema like this to simplify our example. In a real-world application, you should have a proper migration system whose role is to make sure your database schema is in sync. We'll learn how to set one up for SQLAlchemy later in the chapter.

Making insert queries

Now we're ready to make queries! Let's start with the `INSERT` queries to create new rows in our database. In the following example, you can view an implementation of an endpoint to create a new post:

app.py

```
@app.post("/posts", response_model=PostDB, status_code=status.HTTP_201_CREATED)
async def create_post(
    post: PostCreate, database: Database = Depends(get_database)
) -> PostDB:
    insert_query = posts.insert().values(post.dict())
    post_id = await database.execute(insert_query)
```

```
post_db = await get_post_or_404(post_id, database)

return post_db
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

You shouldn't be surprised by the look of it: it's a POST endpoint that accepts a payload following the `PostCreate` model. It also injects the database thanks to our `get_database` dependency.

Interesting things begin in the body of the function:

- On the first line, we build our `INSERT` query. Rather than writing SQL queries by hand, we rely on the SQLAlchemy expression language, which consists of chained method calls. Under the hood, SQLAlchemy will build a proper SQL query for our database engine. This is one of the greatest benefits of such libraries: since it produces the SQL query for you, you won't have to modify your source code if you change your database engine.
- This query is built directly from the `posts` object, which is the `Table` instance that we defined earlier. By using this object, SQLAlchemy directly understands that the query concerns this table and builds the SQL accordingly.
- We start by calling the `insert` method. Then, we move ahead with the `values` method. This simply accepts a dictionary that associates the names of the columns with their values. Hence, we just need to call `dict()` on our Pydantic object. This is why it's important that our model matches the database schema.
- On the second line, we'll actually perform the query. Thanks to `database`, we can execute it asynchronously. For an `insert` query, we'll use the `execute` method, which expects the query in an argument.

An `INSERT` query will return the `id` of the newly inserted row. This is very important because, since we allow the database to automatically increment this identifier, we don't know the `id` of our new post beforehand.

In fact, we need it to retrieve this new row from the database afterward. By doing this, we ensure we have an exact representation of the current object in the database before returning it in the response. For this, we use the `get_post_or_404` function, which we'll talk about next.

Making select queries

Now that we can insert new data into our database, we must be able to read it! Typically, you'll have two kinds of read endpoints in your API: one to list objects and one to get a single object.

Let's start with the endpoint to list our blog posts. You can view it in the following example:

app.py

```
@app.get("/posts")
async def list_posts(
    pagination: Tuple[int, int] = Depends(pagination),
    database: Database = Depends(get_database),
) -> List[PostDB]:
    skip, limit = pagination
    select_query = posts.select().offset(skip).limit(limit)
    rows = await database.fetch_all(select_query)

    results = [PostDB(**row) for row in rows]

    return results
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

Once again, making a query is a two-step operation: first, we build the query thanks to the SQLAlchemy query language. Then, we execute it asynchronously using `database`. In this case, we perform a `SELECT` query using the corresponding method on the `posts` table. Notice how we use the `OFFSET` and `LIMIT` clauses to paginate our list of posts using the variables provided by the pagination dependency. It's the same dependency that we defined in *Chapter 5, Dependency Injections in FastAPI*.

Then, we execute this query with the `fetch_all` method of `database`. This method will return a list of rows that match our query.

Each row is returned in the form of a dictionary that associates column names and their values. Therefore, for each of them, we simply have to instantiate them back to a `PostDB` model by unpacking the dictionary.

The other typical endpoint in a REST API is to get a single object. In the following example, you can see how we implemented this endpoint to retrieve a single post:

app.py

```
@app.get("/posts/{id}", response_model=PostDB)
async def get_post(post: PostDB = Depends(get_post_or_404)) ->
    PostDB:
    return post
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

Predictably, this is a GET endpoint that accepts an `id` in the path parameter. The implementation itself is very light. Indeed, since the logic of retrieving a post by its `id` or raising a 404 error if it doesn't exist will be reused many times, it makes sense to put it in a dependency, `get_post_or_404`. You can view its implementation in the following example:

app.py

```
async def get_post_or_404(
    id: int, database: Database = Depends(get_database)
) -> PostDB:
    select_query = posts.select().where(posts.c.id == id)
    raw_post = await database.fetch_one(select_query)

    if raw_post is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)

    return PostDB(**raw_post)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

Once again, we start by building a SQL query. This time, we have a `WHERE` clause, which only retrieves the row for the `id` we need. The clause itself might look strange.

The first part is to set the actual column we want to compare. Each column is accessible via its name from the `c` attribute of the table object, that is, `posts.c.id`.

Then, we use the equality operator to compare with our actual `id` variable. It looks like a standard comparison that would result in a Boolean, not a SQL statement! In a general Python context, it would. However, SQLAlchemy developers have done something clever here: they overloaded the standard operators so that they produce SQL expressions instead of comparing objects. This is exactly what we saw in the *Magic methods* section of *Chapter 2, Python Programming Specificities*.

Then, we simply call `fetch_one` on the database object. It's a convenient shortcut when we only expect one row at most.

Two things can happen: if no row matches our query, the result is `None`. In this case, we can raise a 404 error. Otherwise, we get the data in the form of a dictionary. All we have to do is to instantiate it back into a `PostDB` model.

Dependencies are like functions

In our POST endpoint, we used `get_post_or_404` as a regular function to retrieve our newly created blog post. This is perfectly okay: dependencies don't have hidden or magic logic inside, so you can reuse them at will. The only thing to remember is that you have to provide every argument manually since you are outside of the dependency injection context.

Making update and delete queries

Finally, let's examine how to update and delete rows in our database. The main difference is how you build the query using SQLAlchemy expressions, but the rest of the implementation is always the same.

In the following example, let's take a look at how to update a blog post:

app.py

```
@app.patch("/posts/{id}", response_model=PostDB)
async def update_post(
    post_update: PostPartialUpdate,
    post: PostDB = Depends(get_post_or_404),
    database: Database = Depends(get_database),
) -> PostDB:
    update_query = (
```

```
posts.update()  
    .where(posts.c.id == post.id)  
    .values(post_update.dict(exclude_unset=True))  
)  
post_id = await database.execute(update_query)  
  
post_db = await get_post_or_404(post_id, database)  
  
return post_db
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

In this case, we start with an UPDATE statement. Upon this, we add a WHERE clause to only match the post we want to update. Finally, we set the values we want to update in the form of a dictionary. As we explained in *Chapter 4, Managing pydantic Data Models in FastAPI*, since we are doing a partial update here, you can see that we use the `exclude_unset` option to only get the values to update.

Deleting an object is not very different, as you can see in the following example:

app.py

```
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_  
CONTENT)  
async def delete_post(  
    post: PostDB = Depends(get_post_or_404), database: Database  
    = Depends(get_database)  
):  
    delete_query = posts.delete().where(posts.c.id == post.id)  
    await database.execute(delete_query)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy/app.py>

It mainly consists of a DELETE statement followed by the adequate WHERE clause.

Now you know how to perform the most common SQL queries using the SQLAlchemy expression language and *databases*. We recommend that you go through the SQLAlchemy expression language tutorial to learn about all the features and the more advanced usage of this powerful tool. You can find the official documentation at <https://docs.sqlalchemy.org/en/13/core/tutorial.html>.

Adding relationships

As we mentioned at the beginning of this chapter, relational databases are all about data and its relationships. Quite often, you'll need to create entities that are linked to others. For example, in a blog application, comments are linked to the post they relate to. In this section, we'll examine how you can set up such relationships with SQLAlchemy. Since it's very close to SQL, you'll discover that there's nothing truly surprising about it.

First, we need to define the table for the comments, which has a foreign key toward the `posts` table. You can view its definition in the following example:

models.py

```
comments = sqlalchemy.Table(
    "comments",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_
key=True, autoincrement=True),
    sqlalchemy.Column(
        "post_id", sqlalchemy.ForeignKey("posts.id",
ondelete="CASCADE"), nullable=False
    ),
    sqlalchemy.Column("publication_date", sqlalchemy.
DateTime(), nullable=False),
    sqlalchemy.Column("content", sqlalchemy.Text(),
nullable=False),
)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy_relationship/models.py

The important point here is the `post_id` column, which is of the `ForeignKey` type. This is a special type that tells SQLAlchemy to automatically handle the type of the column and the associated constraint. We simply have to give the table and column names it refers to. Note that we can also specify the `ON DELETE` action.

We won't go into the details of the Pydantic models for the comments since they are quite straightforward. However, we want to highlight a new model we created for the posts, that is, `PostPublic`. This is shown in the following example:

models.py

```
class PostPublic(PostDB):
    comments: List[CommentDB]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy_relationship/models.py

Here, you can see that we added a `comments` attribute, which is a list of `CommentDB`. Indeed, in a REST API, there are some cases where it makes sense to automatically retrieve the associated objects of an entity. Here, it'll be convenient to get the comments of a post in a single request. We'll use this model when getting a single post to *serialize the comments along with the post data*.

Now, we'll implement an endpoint to create a new comment. This is shown in the following example:

app.py

```
@app.post("/comments", response_model=CommentDB, status_code=status.HTTP_201_CREATED)
async def create_comment(
    comment: CommentCreate, database: Database = Depends(get_database)
) -> CommentDB:
    select_post_query = posts.select().where(posts.c.id == comment.post_id)
    post = await database.fetch_one(select_post_query)

    if post is None:
        raise HTTPException(
```

```

        status_code=status.HTTP_400_BAD_REQUEST,
        detail=f"Post {id} does not exist"
    )

    insert_query = comments.insert().values(comment.dict())
    comment_id = await database.execute(insert_query)

    select_query = comments.select().where(comments.c.id ==
comment_id)
    raw_comment = cast(Mapping, await database.fetch_
one(select_query))

    return CommentDB(**raw_comment)

```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy_relationship/app.py

Note that the endpoint parameters and most of the implementations are very close to the create post endpoint. The only difference here is the first part of the function logic where we check for the existence of the post before proceeding with the comment creation. This is important because, since the end user can send any post ID, we could have a situation where we try to create a comment for a post that doesn't exist, which could cause a constraint error at the database level. This is why we are trying to get the post first and then show a clear error to prevent this situation.

Earlier, we mentioned that we wanted to retrieve a post and its comments at the same time. To do this, we'll have to make a second query to retrieve the comments and then merge all the data together in a `PostPublic` instance. We added this logic in the `get_post_or_404` dependency, as you can see in the following example:

app.py

```

async def get_post_or_404(
    id: int, database: Database = Depends(get_database)
) -> PostPublic:
    select_post_query = posts.select().where(posts.c.id == id)
    raw_post = await database.fetch_one(select_post_query)

    if raw_post is None:

```

```
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)

    select_post_comments_query = comments.select().
    where(comments.c.post_id == id)

    raw_comments = await database.fetch_all(select_post_comments_query)

    comments_list = [CommentDB(**comment) for comment in raw_comments]

    return PostPublic(**raw_post, comments=comments_list)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy_relationship/app.py

Here, you can see that we simply add a `fetch_all` query with the correct `WHERE` statement to collect the comments associated with the post. Then, we only have to transform them into a list of `CommentDB` and set it during `PostPublic` initialization.

Why not make a JOIN query?

Admittedly, by making a `JOIN` query, we could retrieve the post and the comments data in one query instead of two. The problem with `JOIN` queries is that they return as many rows as there are comments, all concatenated with the post data. While this is possible, clever logic is required to separate the post data and create a list of comments. For the simplicity of this example, we have chosen to perform two queries.

Essentially, that's it for working with relationships with `SQLAlchemy`. You can see that, since we are very close to `SQL`, it's up to you to build the right queries to shape the data as needed and resolve the relations.

Setting up a database migration system with Alembic

When developing an application, you'll likely make changes to your database schema to add new tables, add new columns, or modify existing ones. Of course, if your application is already in production, you don't want to erase all your data to recreate the schema from scratch: you want them to be migrated to the new schema. Tools for this task have been developed, and in this section, we'll learn how to set up *Alembic*, from the creators of `SQLAlchemy`. Let's install this library:

```
$ pip install alembic
```

Once this has been completed, you'll have access to the `alembic` command to manage this migration system. When starting a new project, the first thing to do is to initialize the migration environment, which includes a set of files and directories where Alembic will store its configuration and migration files. At the root of your project, run the following command:

```
$ alembic init alembic
```

This will create a directory, named `alembic`, at the root of your project. You can view the result of this command in the example repository shown in *Figure 6.5*:

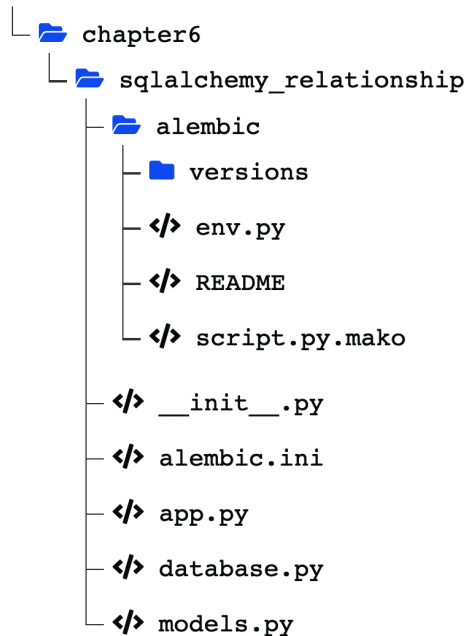


Figure 6.5 – The Alembic migration environment structure

This folder will contain all the configurations for your migrations and your migration scripts themselves. It should be committed along with your code to keep a record of the versions of those files.

Additionally, note that it created an `alembic.ini` file, which contains all the configuration options of Alembic. We'll review two important settings of this file: `script_location` and `sqlalchemy.url`. You can view the first one in the following example:

alembic.ini

```
# path to migration scripts
```

```
script_location = chapter6/sqlalchemy_relationship/alembic
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy\_relationship/alembic.ini
```

This setting expects the path of the `alembic` directory containing the migration files. In most of your projects, this will just be `alembic`, because it'll simply be at the root of your project. Here, since we have several projects in our example repository, we had to set a sub-folder path.

The second important option is `sqlalchemy.url`, which you can view in the following example:

alembic.ini

```
sqlalchemy.url = sqlite:///chapter6_sqlalchemy_relationship.db
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy\_relationship/alembic.ini
```

Predictably, this is the connection string of your database that will receive the migration queries. It follows the same convention that we saw earlier. Here, we set our SQLite database.

Next, we'll focus on the `env.py` file. This is a Python script containing all the logic executed by Alembic to initialize the migration engine and execute the migrations. Being a Python script allows us to finely customize the execution of Alembic. For the time being, we'll keep the default one except for one thing: we'll import our `metadata` object. You can view this in the following example:

env.py

```
from chapter6.sqlalchemy_relationship.models import metadata

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
fileConfig(config.config_file_name)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
target_metadata = metadata
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy_relationship/alembic/env.py

By default, the file defines a variable named `target_metadata`, which is set to `None`. Here, we changed it so that it refers to the `metadata` object that we just imported from our `models` module. But why do we do that? Well, remember that `metadata` is a SQLAlchemy object that contains all the table definitions. By providing it to Alembic, the migration system will be able to *automatically generate the migration scripts* just by looking at your schema! This way, you won't have to write them from scratch.

When you have made changes to your database schema, you can run the following command to generate a new migration script:

```
$ alembic revision --autogenerate -m "Initial migration"
```

It'll create a new script in the version's directory with the commands reflecting your schema changes. You can view how it looks in the following example:

a12742852e8c_initial_migration.py

```
def upgrade():
    # ### commands auto generated by Alembic - please adjust!
    ###
    op.create_table(
        "posts",
        sa.Column("id", sa.Integer(), autoincrement=True,
            nullable=False),
        sa.Column("publication_date", sa.DateTime(),
            nullable=False),
        sa.Column("title", sa.String(length=255),
            nullable=False),
        sa.Column("content", sa.Text(), nullable=False),
        sa.PrimaryKeyConstraint("id"),
    )
    op.create_table(
        "comments",
        sa.Column("id", sa.Integer(), autoincrement=True,
            nullable=False),
        sa.Column("post_id", sa.Integer(), nullable=False),
        sa.Column("publication_date", sa.DateTime(),
            nullable=False),
        sa.Column("content", sa.Text(), nullable=False),
        sa.ForeignKeyConstraint(["post_id"], ["posts.id"],
            ondelete="CASCADE"),
        sa.PrimaryKeyConstraint("id"),
    )
    # ### end Alembic commands ###

def downgrade():
    # ### commands auto generated by Alembic - please adjust!
    ###
    op.drop_table("comments")
```

```
op.drop_table("posts")
# ### end Alembic commands ###
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/sqlalchemy_relationship/alembic/versions/a12742852e8c_initial_migration.py

Here, we have the required operations to create our posts and comments table, with all of their columns and constraints. Notice that we have two functions: `upgrade` and `downgrade`. The first one is used to *apply the migration* and the second one is used to *roll it back*. This is very important because if something goes wrong during the migration, or if you need to revert to an older version of your application, you'll be able to do so without breaking your data.

Autogenerate doesn't detect everything

Bear in mind that, even though autogeneration is very helpful, it's not always accurate, and, sometimes, it's not able to detect ambiguous changes. For example, if you rename a column, it will delete the old one and create another. As a result, the data for this column will be lost! This is why you should always carefully review the migration scripts and make the required changes for edge cases like this.

Finally, you can apply the migrations to your database using the following command:

```
$ alembic upgrade head
```

This will run all the migrations that have not yet been applied to your database until the latest. It's interesting to know that, in the process, Alembic creates a table in your database so that it can remember all the migrations it has applied: this is how it detects which scripts to run.

Generally speaking, you should be *extremely careful* when you run such commands on your database, especially on a production one. Very bad things can happen if you make a mistake, and you can lose precious data. You should always test your migrations in a test environment and have fresh and working backups before running them on your production database.

This is a very quick introduction to Alembic and its powerful migration system. We strongly encourage you to go through its documentation to understand all of its mechanisms, especially regarding migration script operations. Please refer to <https://alembic.sqlalchemy.org/en/latest/index.html>.

That's it for the SQLAlchemy part of this chapter! If you are used to relational databases and SQL, you shouldn't have been too surprised by its usage; it's very close to SQL. However, sometimes, it's quicker and more convenient to move slightly away from SQL and let libraries do the querying for us. This is exactly what ORM is for.

Communicating with a SQL database with Tortoise ORM

When dealing with relational databases, you might wish to abstract away the SQL concepts and only deal with proper objects from the programming language. That's the main motivation behind ORM tools. In this section, we'll examine how to work with *Tortoise ORM*, which is a modern and asynchronous ORM that fits nicely within a FastAPI project. It's greatly inspired by the Django ORM; so, if you've ever worked with it, you'll probably be on familiar ground.

As usual, the first step is to install the library using the following command:

```
$ pip install tortoise-orm
```

If you need drivers for database engines such as PostgreSQL or MySQL, you can install them, as explained in the documentation at https://tortoise-orm.readthedocs.io/en/latest/getting_started.html#installation. We're now ready to work!

Creating database models

The first step is to create the Tortoise model for your entity. This is a Python class whose attributes represent the columns of your table. This class will provide you static methods in which to perform queries, such as retrieving or creating data. Moreover, the actual entities of your database will be instances of this class, giving you access to its data like any other object. Under the hood, the role of Tortoise is to make the link between this Python object and the row in the database. Let's take a look at the definition of our blog post model in the following example:

models.py

```
class PostTortoise(Model):
    id = fields.IntField(pk=True, generated=True)
    publication_date = fields.DatetimeField(null=False)
    title = fields.CharField(max_length=255, null=False)
```

```
content = fields.TextField(null=False)
```

```
class Meta:
```

```
    table = "posts"
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/models.py>

Our model is a class that is inheriting from the `tortoise.models.Model` base class. Each field (or column) is an instance of a class corresponding to the type of the field. Each one has its own set of arguments to finely tune the definition in the database. For example, our `id` field is a primary key that is automatically generated. We won't go through every field's class, but you can find the complete list in the official Tortoise documentation at <https://tortoise-orm.readthedocs.io/en/latest/fields.html>.

Notice that we also have a sub-class called `Meta`, which allows us to set some options for our table. Here, the `table` attribute allows us to control the name of the table.

If you look at the code above the table definition, you'll see that we have also defined the corresponding Pydantic models for our post entity. They will be used by FastAPI to perform data validation and serialization. As you can see in the following example, we added a `Config` sub-class and set an attribute called `orm_mode`:

models.py

```
class PostBase(BaseModel):
```

```
    title: str
```

```
    content: str
```

```
    publication_date: datetime = Field(default_
factory=datetime.now)
```

```
class Config:
```

```
    orm_mode = True
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/models.py>

This option will allow us to transform an ORM object instance into a Pydantic object instance. This is essential because FastAPI is designed to work with Pydantic models, not ORM models.

Here, we hit what is maybe the most confusing part about working with FastAPI and an ORM: we'll have to work with both ORM objects and Pydantic models and find ways to transform them back and forth.

If you refer to the Tortoise documentation, you'll find that it tries to solve this by providing tools to automatically generate Pydantic models from Tortoise ones. We won't show this approach in this book because it comes with some pitfalls and is less flexible than pure Pydantic models. Nevertheless, once you are confident with the concepts we are showing here, we encourage you to try this approach and see if it fits your needs.

Setting up the Tortoise engine

Now that we have our model ready, we have to configure the Tortoise engine to set the database connection string and the location of our models. To do this, Tortoise comes with a utility function for FastAPI that does all the required tasks for you. In particular, it automatically adds event handlers to open and close the connection at startup and shutdown; this is something we had to do by hand with SQLAlchemy.

You can see what it looks like in the following example:

app.py

```
TORTOISE_ORM = {
    "connections": {"default": "sqlite://chapter6_tortoise.db"},
    "apps": {
        "models": {
            "models": ["chapter6.tortoise.models"],
            "default_connection": "default",
        },
    },
}

register_tortoise(
    app,
    config=TORTOISE_ORM,
    generate_schemas=True,
    add_exception_handlers=True,
)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

As you can see, we put the main configuration options in a variable named `TORTOISE_ORM`. Let's review its different fields:

- The `connections` key contains a dictionary associating a database alias to a connection string, which gives access to your database. It follows the standard convention, as explained in the documentation at https://tortoise-orm.readthedocs.io/en/latest/databases.html?highlight=db_url#db-url.

In most projects, you'll probably have one database named `default`, but it allows you to set several databases if needed.

- In the `apps` key, you'll be able to declare all your modules containing your Tortoise models. The first key just below `apps`, that is, `models`, will be the prefix with which you'll be able to refer to the associated models. You can name it how you want, but if you place all your models under the same scope, then `models` is a good candidate. This prefix is especially important when defining foreign keys. For example, with this configuration, our `PostTortoise` model can be referred to by the name `models.PostTortoise`. It's not the actual path to your module.

Underneath it, you have to list all the modules containing your models. Additionally, we set the corresponding database connection with the alias we defined earlier.

Then, we call the `register_tortoise` function that'll take care of setting up Tortoise for FastAPI. Let's explain its arguments:

- The first one is your FastAPI app instance.
- Then, we have the configuration that we defined earlier.
- Setting `generate_schemas` to `True` will automatically create the table's schema in the database. Otherwise, our database will be empty and we won't be able to insert any rows.

While this is useful for testing purposes, in a real-world application, you should have a proper migration system whose role is to make sure your database schema is in sync. We'll examine how to set one up for Tortoise later in the chapter.

- Finally, the `add_exception_handlers` option adds custom exception handlers to FastAPI, allowing you to nicely catch Tortoise errors and return proper error responses.

And that's all! Always make sure that you call this function at the end of your application file, to ensure everything has been correctly imported. Apart from that, Tortoise handles everything for us. We're now ready to go!

Creating objects

Let's start by inserting new objects inside our database. The main challenge is to transform the Tortoise object instance into a Pydantic model. Let's review this in the following example:

app.py

```
@app.post("/posts", response_model=PostDB, status_code=status.HTTP_201_CREATED)
async def create_post(post: PostCreate) -> PostDB:
    post_tortoise = await PostTortoise.create(**post.dict())

    return PostDB.from_orm(post_tortoise)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

Here, we have our POST endpoint, which accepts our `PostCreate` model. The core logic consists then of two operations.

First, we create the object in the database. We directly use the `PostTortoise` class and its static `create` method. Conveniently, it accepts a dictionary that maps fields to their values, so we just have to call `dict` on our input object. Of course, this operation is natively asynchronous!

As a result, we get an instance of a `PostTortoise` object. This is why the second operation we need to perform is to transform it into a Pydantic model. To do this, we use the `from_orm` method, which is available because we enabled `orm_mode`. We get a proper `PostDB` instance, which we can return directly.

Can we return a PostTortoise object directly?

Technically, yes, we can. In the case of a Tortoise model, it implements the magic methods to be transformed into a dictionary, which is the last fallback of FastAPI when it doesn't recognize the object you have returned. However, doing this would deprive us of all the goodness of using Pydantic models, such as field exclusion or automatic documentation. This is why we recommend here that you always go back to a Pydantic model.

Here, you can see that the implementation is quite straightforward. Now, let's retrieve this data!

Getting and filtering objects

Usually, a REST API provides two types of endpoints to read data: one to list objects and one to get a specific object. This is exactly what we'll review next!

In the following example, you can see how we implemented the endpoint to list objects:

app.py

```
@app.get("/posts")
async def list_posts(pagination: Tuple[int, int] =
    Depends(pagination)) -> List[PostDB]:
    skip, limit = pagination
    posts = await PostTortoise.all().offset(skip).limit(limit)

    results = [PostDB.from_orm(post) for post in posts]

    return results
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

Once again, this is an operation in two steps: first, we retrieve Tortoise objects using the query language. Notice that we use the `all` method, which gives us every object in the table. Additionally, we're able to apply our pagination parameters through `offset` and `limit`.

Then, we have to transform this list of `PostTortoise` objects into a list of `PostDB` objects. Again, thanks to `from_orm` and a list comprehension, we can do this very easily.

Now, in the following example, we'll take a look at the endpoint to retrieve a single post:

app.py

```
@app.get("/posts/{id}", response_model=PostDB)
async def get_post(post: PostTortoise = Depends(get_post_or_404)) -> PostDB:
    return PostDB.from_orm(post)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

This is a simple GET endpoint that expects the ID of the post in the path parameter. The implementation is itself very light: we just have to transform our `PostTortoise` object into a `PostDB`. Most of the logic is in the `get_post_or_404` dependency, which we'll reuse often in our application. The following example shows its implementation:

app.py

```
async def get_post_or_404(id: int) -> PostTortoise:
    return await PostTortoise.get(id=id)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

The role of this dependency is to take the `id` in the path parameter and retrieve a single object from the database that corresponds to this identifier. The `get` method is a convenient shortcut for this: if no matching record is found, it raises the `DoesNotExist` exception. If there is more than one matching record, it raises `MultipleObjectsReturned`.

You might be wondering where our exception handler is to raise a proper 404 error. In fact, it's already there, at a global level! Remember that we set up Tortoise with the `add_exception_handlers` option: under the hood, it adds a handler that automatically catches `DoesNotExist` and builds a proper 404 error. So, we don't have to do anything more!

Updating and deleting objects

We'll finish by showing you how to update and delete existing objects. The logic is always the same; we just have to adapt the methods we call on our Tortoise object.

In the following example, you can view the implementation of the update endpoint:

app.py

```
@app.patch("/posts/{id}", response_model=PostDB)
async def update_post(
    post_update: PostPartialUpdate, post: PostTortoise =
    Depends(get_post_or_404)
) -> PostDB:
    post.update_from_dict(post_update.dict(exclude_unset=True))
    await post.save()

    return PostDB.from_orm(post)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

Here, the main point of attention is that we'll operate directly on the post we want to modify. This is one of the key aspects when working with ORM: entities are objects that can be modified as you wish. When you are happy with the data, you can persist it in the database. This is exactly what we do here: we get a fresh representation of our post thanks to `get_post_or_404` and apply the `update_from_dict` utility method to change the fields that we want. Then, we can persist the changes in the database using `save`.

The same concept is applied when you wish to delete an object: when you have an instance, you can call `delete` to physically remove it from the database. You can view this in action in the following example:

app.py

```
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_
CONTENT)
async def delete_post(post: PostTortoise = Depends(get_post_
or_404)):
    await post.delete()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise/app.py>

That's almost it for the basics of working with Tortoise ORM. Of course, we only covered the most basic queries, but you can do far more complex things. You can find a thorough overview of the query language in the official documentation at <https://tortoise-orm.readthedocs.io/en/latest/query.html#query-api>.

Adding relationships

Now, let's take a look at how to work with relationships. Once again, we'll examine how to implement comments that are linked to posts. One of the main tasks of Tortoise, and ORM in general, is to ease the process of working with related entities, by automatically making the required JOIN queries and instantiating sub-objects. However, once again, there are some things that we need to take care of to make sure everything works smoothly with Pydantic.

We'll begin by creating a model for our comment entity, as shown in the following example:

models.py

```
class CommentTortoise(Model):
    id = fields.IntField(pk=True, generated=True)
    post = fields.ForeignKeyField(
        "models.PostTortoise", related_name="comments",
        null=False
    )
    publication_date = fields.DateTimeField(null=False)
    content = fields.TextField(null=False)

    class Meta:
        table = "comments"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise_relationship/models.py

The main point of interest here is the `post` field, which is purposely defined as a foreign key. The first argument is the reference to the associated model. Notice that we use the `models` prefix; this is the same one we defined in the Tortoise configuration that we saw earlier. Additionally, we set the `related_name`. This is a typical and convenient feature of ORM. By doing this, we'll be able to get all the comments of a given post simply by accessing its `comments` property. The action of querying the related comments, therefore, becomes completely *implicit*.

In the next example, we'll look at the base Pydantic model for a comment, `CommentBase`:

models.py

```
class CommentBase(BaseModel):
    post_id: int
    publication_date: datetime = Field(default_factory=datetime.now)
    content: str

class Config:
    orm_mode = True
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise_relationship/models.py

Here, you can see that we have defined a `post_id` attribute. This attribute will be used in the request payload to set the post that we want to attach this new comment to. When you provide this attribute to Tortoise, it automatically understands that you are referring to the identifier of the foreign key field, called `post`.

In a REST API, sometimes, it makes sense to automatically retrieve the associated objects of an entity in one request. Here, we'll ensure that the comments of a post are returned in the form of a list along with the post data. To do this, we introduce a new Pydantic model, `PostPublic`. You can view this in the following example:

models.py

```
class PostPublic(PostDB):
    comments: List[CommentDB]
```

```
@validator("comments", pre=True)
def fetch_comments(cls, v):
    return list(v)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise_relationship/models.py

Predictably, we simply added a `comments` attribute, which is a list of `CommentDB`. However, here, you can see something unexpected: a validator for this attribute. Earlier, we mentioned that thanks to Tortoise, we can retrieve the comments of a post by simply doing `post.comments`. This is convenient, but this attribute is not directly a list of data: it's a query set object. If we don't do anything, then, when we try to transform the ORM object into a `PostPublic`, Pydantic will try to parse this query set and fail. However, calling `list` on this query set forces it to output the data. That is the purpose of this validator. Notice that we set it with `pre=True` to make sure it's called before the built-in Pydantic validation.

We'll now implement an endpoint to create a new comment. This is shown in the following example:

app.py

```
@app.post("/comments", response_model=CommentDB, status_code=status.HTTP_201_CREATED)
async def create_comment(comment: CommentBase) -> CommentDB:
    try:
        await PostTortoise.get(id=comment.post_id)
    except DoesNotExist:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Post {id} does not exist"
        )

    comment_tortoise = await CommentTortoise.create(**comment.dict())

    return CommentDB.from_orm(comment_tortoise)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise_relationship/app.py

Most of the logic is very similar to the create post endpoint. The main difference is that we first check for the existence of the post before proceeding with the comment creation. Indeed, we want to avoid the foreign key constraint error that could occur at the database level and show a clear and helpful error message to the end user instead.

As we mentioned earlier, our objective is to output the comments when retrieving a single post. To do this, we made a small change to the `get_post_or_404` dependency, as follows:

app.py

```
async def get_post_or_404(id: int) -> PostTortoise:
    try:
        return await PostTortoise.get(id=id).prefetch_
            related("comments")
    except DoesNotExist:
        raise HTTPException(status_code=status.HTTP_404_NOT_
            FOUND)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise_relationship/app.py

The only difference here is that we called the `prefetch_related` method on our query. By passing in the name of the related entities, it allows you to preload them upfront when getting the main object. By default, Tortoise is lazy and doesn't make the additional query. In our case, it's not just an optimization: it's important to ensure our code is working. Indeed, if our validator tries to call `list` on a query set that hasn't been prefetched, it'll raise an error. This is because of the asynchronous nature of the ORM: you have to retrieve the data asynchronously, with a proper `await` statement, before you can operate over the data normally.

Other than that, there is nothing more you need to do. The key takeaway here is that you have to pay attention when trying to work with relationships and make sure you resolve them correctly before feeding them to Pydantic.

Setting up a database migration system with Aerich

In the *Setting up a database migration system with Alembic* section of this chapter, we already mentioned the need for a database migration system. When you make changes to your database schema, you want to migrate your existing data in production in a safe and reproducible manner. In this section, we'll demonstrate how to install and configure Aerich, which is a database migration tool from the creators of Tortoise. As usual, we'll start by installing the library:

```
$ pip install aerich
```

Once this is done, you'll have access to the `aerich` command to manage this migration system.

The first thing you need to do is declare the Aerich models in your Tortoise configuration. Indeed, Aerich stores some migration state information in your database. You can view what the configuration looks like in the following example:

app.py

```
TORTOISE_ORM = {
    "connections": {"default": "sqlite://chapter6_tortoise_
relationship.db"},
    "apps": {
        "models": {
            "models": ["chapter6.tortoise_relationship.models",
"aerich.models"],
            "default_connection": "default",
        },
    },
}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/tortoise_relationship/app.py

Then, you can initialize the migration environment, which is a set of files and directories where Aerich will store its configuration and migration files. The command looks like this:

```
$ aerich init -t chapter6.tortoise_relationship.app.TORTOISE_ORM
```

The `-t` option should refer to the dotted path of your `TORTOISE_ORM` configuration variable. This is how Aerich is able to retrieve your database connection information and the definition of your models. Then, you have to call the following command:

```
$ aerich init-db
```

Following this, your project structure should look similar to the one shown in *Figure 6.5*:

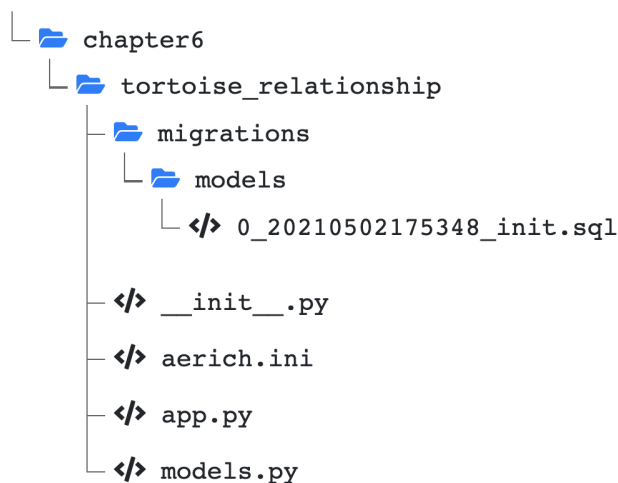


Figure 6.6 – The Aerich migration environment structure

The `migrations` folder will contain all of the migration scripts. Notice that it creates a sub-directory for each of the "apps" defined in the configuration. As you can see, we have a first migration script that creates all the tables that have already been defined.

It also adds the `aerich.ini` configuration file, which essentially sets the path to your configuration variable and `migrations` folder.

To apply the migrations to your database, simply run the following command:

```
$ aerich upgrade
```

During the life of your project, when you have made changes to your table's schema, you'll have to generate new migration scripts to reflect the changes. This is done quite easily using the following command:

```
$ aerich migrate --name added_new_tables
```

The `--name` option allows you to set a name for your migration. It will automatically generate a new migration file that reflects your changes.

Aerich migration scripts are not cross-database compatible

Contrary to Alembic, Aerich doesn't abstract migration operations through cross-compatible Python scripts. Instead, it directly generates SQL files that are compatible with the engine you are working with. Since there are significant differences between the various SQL implementations, you can't work, for example, on a SQLite database during development and have a PostgreSQL for production: the migration scripts generated locally wouldn't work on your production server. This is why you should have the same database engine both in local and in production.

Just as with any automated migration system, you should always review the generated scripts to make sure they correctly reflect your changes and that you don't lose data in the process. Always test your migrations in a test environment and have fresh and working backups before running them in production.

That's it for this introduction to Tortoise ORM. If you have ever used an ORM before, you should be already confident with it. The main challenge to tackle with FastAPI is to make it work together with Pydantic models to get the benefit of both worlds. We'll now leave the world of relational databases to explore how we can work with a document-oriented database, MongoDB.

Communicating with a MongoDB database using Motor

As we mentioned at the beginning of this chapter, working with a document-oriented database, such as MongoDB, is quite different from a relational database. First and foremost, you don't need to configure a schema upfront: it follows the structure of the data that you insert into it. In the case of FastAPI, it makes our life slightly easier since we'll only have to work with Pydantic models. However, there are some subtleties around the document identifiers that we need to take into account. We'll review this next.

To begin, we'll install Motor, which is a library that is used to communicate asynchronously with MongoDB and is officially supported by the MongoDB organization. You can run the following command:

```
$ pip install motor
```

Once this is done, we can start working!

Creating models compatible with MongoDB ID

As we mentioned in the introduction to this section, there are some difficulties with the identifiers that MongoDB uses to store documents. Indeed, by default, MongoDB assigns every document an `_id` property that acts as a unique identifier in a collection. This causes two issues:

- In a Pydantic model, if a property starts with an underscore, it's considered to be private and, thus, is not used as a data field for our model.
- `_id` is encoded as a binary object, called `ObjectId`, instead of a simple integer or string. It's usually represented in the form of a string such as `608d1ee317c3f035100873dc`. This type of object is not supported out of the box by Pydantic or FastAPI.

This is why we'll need some boilerplate code to ensure those identifiers work with Pydantic and FastAPI. To begin, in the following example, we have created a `MongoBaseModel` base class that takes care of defining the `id` field:

models.py

```
class MongoBaseModel(BaseModel):
    id: PyObjectId = Field(default_factory=PyObjectId, alias="_id")

class Config:
    json_encoders = {ObjectId: str}
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/models.py>

First, we need to define an `id` field, which is of type `PyObjectId`. This is a custom type that has been defined in the preceding code. We won't go into the details of its implementation, but just know that it's a class that makes `ObjectId` a compatible type for Pydantic. We define this same class as a default factory for this field. Interestingly, that kind of identifier allows us to generate them on the client side, contrary to traditional auto-incremented integers of relational databases, which could be useful in some cases.

The most interesting argument is `alias`. It's a Pydantic option allowing us to *change the name of the field during serialization*. In this example, when we call the `dict` method on one instance of `MongoBaseModel`, the identifier will be set on the `_id` key; this is the name expected by MongoDB. That solves the first issue.

Then, we add the `Config` sub-class and set the `json_encoders` option. By default, Pydantic is completely unaware of our `PyObjectId` type, so it won't be able to correctly serialize it to JSON. This option allows us to *map custom types with a function that will be called to serialize them*. Here, we simply transform it into a string (it works because `ObjectId` implements the `__str__` magic method). That solves the second issue for Pydantic.

Our base model for Pydantic is complete! We can now use it as a base class instead of `BaseModel` for our actual data models. Notice, however, that the `PostPartialUpdate` doesn't inherit from it. Indeed, we don't want the `id` field in this model; otherwise, a `PATCH` request might be able to replace the ID of the document, which could lead to weird issues.

Connecting to a database

Now that our models are ready, we can set up the connection with a MongoDB server. This is quite easy and only involves a class instantiation, as shown in the following example:

app.py

```
motor_client = AsyncIOMotorClient("mongodb://localhost:27017")
# Connection to the whole server

database = motor_client["chapter6_mongo"] # Single database
instance

def get_database() -> AsyncIOMotorDatabase:
    return database
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

Here, you can see that `AsyncIOMotorClient` simply expects a connection string to your database. Generally, it consists of the scheme, followed by authentication information and the hostname of the database server. You can find an overview of this format in the official MongoDB documentation at <https://docs.mongodb.com/manual/reference/connection-string/>.

However, be careful. Contrary to the libraries we've discussed so far, the client instantiated here is not bound to any database, that is, it's only a connection to a whole server. That's why we need the second line to set the database that we want to work upon directly by its key. It's worth noting that MongoDB doesn't require you to create the database upfront: it'll create it automatically if it doesn't exist.

Then, we create a simple function to return this database instance. We'll use this function as a dependency to retrieve this instance in our path operation functions. We explained the benefits of this pattern in the *Communicating with a SQL database with SQLAlchemy* section.

That's it! We can now make queries to our database!

Inserting documents

We'll start by demonstrating how to implement an endpoint to create posts. Essentially, we just have to insert our Pydantic model that has been transformed into a dictionary:

app.py

```
@app.post("/posts", response_model=PostDB, status_code=status.HTTP_201_CREATED)
async def create_post(
    post: PostCreate, database: AsyncIOMotorDatabase = Depends(get_database)
) -> PostDB:
    post_db = PostDB(**post.dict())
    await database["posts"].insert_one(post_db.dict(by_alias=True))

    post_db = await get_post_or_404(post_db.id, database)

    return post_db
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

Classically, this is a POST endpoint that accepts a payload in the form of a `PostCreate` model. Additionally, we inject the database instance with the dependency we wrote earlier.

In the path operation itself, you can see that we start by instantiating a `PostDB` from the `PostCreate` data. This is usually a good practice if you only have fields in `PostDB` that need to be initialized.

Then, we have the query. To retrieve a collection in our MongoDB database, we simply have to get it by name, like a dictionary. Once again, MongoDB will take care of creating it if it doesn't exist. As you can see, document-oriented databases are much more lightweight regarding schema than relational databases! In this collection, we can call the `insert_one` method to insert a single document. It expects a dictionary to map fields to their values. Therefore, the `dict` method of Pydantic objects is once again our friend. However, here, we see something new: we call it with the `by_alias` argument set to `True`. By default, Pydantic will serialize the object with the real field name, not the alias name. However, we do need the identifier named as `_id` in our MongoDB database. Using this option, Pydantic will use the alias as a key in the dictionary.

To ensure we have a true and fresh representation of our document in the dictionary, we retrieve it back from the database thanks to our `get_post_or_404` function. We'll examine how it works in the next section.

Getting documents

Of course, retrieving the data from the database is an important part of the job of a REST API. Now, we'll demonstrate how to implement two classic endpoints, that is, to list posts and get a single post. Let's start with the first one and take a look at its implementation in the following example:

app.py

```
@app.get("/posts")
async def list_posts(
    pagination: Tuple[int, int] = Depends(pagination),
    database: AsyncIOMotorDatabase = Depends(get_database),
) -> List[PostDB]:
    skip, limit = pagination
    query = database["posts"].find({}, skip=skip, limit=limit)

    results = [PostDB(**raw_post) async for raw_post in query]

    return results
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

The most interesting part is the second line where we define the query. After retrieving the `posts` collection, we call the `find` method. The first argument should be the filtering query, following the MongoDB syntax. Since we want every document, we leave it empty. Then, we have keyword arguments that allow us to apply our pagination parameters.

MongoDB returns us a result in the form of a list of dictionaries, which maps fields to their values. This is why we added a list comprehension construct to transform them back into `PostDB` instances so that FastAPI can serialize them properly.

You might have noticed something quite surprising here: contrary to what we do usually, we didn't wait for the query directly. Instead, we added the `async` keyword to our list comprehension. Indeed, in this case, Motor returns an **asynchronous generator**. It's the asynchronous counterpart of the classic generator. It works in the same way, aside from the `async` keyword we have to add when iterating over it.

Now, let's take a look at the endpoint to retrieve a single post. The following example shows its implementation:

app.py

```
@app.get("/posts/{id}", response_model=PostDB)
async def get_post(post: PostDB = Depends(get_post_or_404)) ->
PostDB:
    return post
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

As you can see, it's a simple GET endpoint that accepts the `id` post as a path parameter. Most of the logic implementation is in the reusable `get_post_or_404` dependency. You can view how it looks like in the next example:

App.py

```
async def get_post_or_404(
    id: ObjectId = Depends(get_object_id),
    database: AsyncIOMotorDatabase = Depends(get_database),
) -> PostDB:
    raw_post = await database["posts"].find_one({"_id": id})
```

```
if raw_post is None:
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)

return PostDB(**raw_post)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

The logic is quite similar to what we saw for the list endpoint. This time, however, we call the `find_one` method with a query to match the post identifier: the key is the name of the document attribute we want to filter on, and the value is the one we are looking for.

This method returns the document in the form of a dictionary or `None` if it doesn't exist. In this case, we raise a proper 404 error.

Finally, we transform it back into a `PostDB` model before returning it.

You might have noticed that we got the `id` through a dependency, `get_object_id`. Indeed, FastAPI will return a string from the path parameter. If we try to make a query with the `id` in the form of a string, MongoDB will not match with the actual binary IDs. That's why we use another dependency that transforms the identifier represented as a string (such as `608d1ee317c3f035100873dc`) to a proper `ObjectId`.

On a side note, here, you have a very nice example of *nested dependencies*: endpoints use the `get_post_or_404` dependency, which itself gets a value from `get_object_id`. You can view the implementation of this dependency in the following example:

app.py

```
async def get_object_id(id: str) -> ObjectId:
    try:
        return ObjectId(id)
    except (errors.InvalidId, TypeError):
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

Here, we simply retrieve the `id` string from the path parameters and try to instantiate it back into an `ObjectId`. If it's not a valid value, we catch the corresponding errors and consider it as a 404 error.

With this, we have solved every challenge posed by the MongoDB identifiers format. Let's now discuss how to update and delete documents.

Updating and deleting documents

We'll now review the endpoints to update and delete documents. The logic is still the same and only involves building the proper query from the request payload.

Let's start with the `PATCH` endpoint, which you can view in the following example:

app.py

```
@app.patch("/posts/{id}", response_model=PostDB)
async def update_post(
    post_update: PostPartialUpdate,
    post: PostDB = Depends(get_post_or_404),
    database: AsyncIOMotorDatabase = Depends(get_database),
) -> PostDB:
    await database["posts"].update_one(
        {"_id": post.id}, {"$set": post_update.dict(exclude_unset=True)}
    )

    post_db = await get_post_or_404(post.id, database)

    return post_db
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

Here, you can see that we use the `update_one` method to update one document. The first argument is the filtering query and the second one is the actual operation to apply to the document. Once again, it follows the MongoDB syntax: the `$set` operation allows us to only modify the fields we want to change by passing the update dictionary.

The DELETE endpoint is even simpler: it's just a single query, as you can see in the following example:

app.py

```
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
async def delete_post(
    post: PostDB = Depends(get_post_or_404),
    database: AsyncIOMotorDatabase = Depends(get_database),
):
    await database["posts"].delete_one({"_id": post.id})
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

The `delete_one` method expects the filtering query as the first argument.

That's it! Of course, here, we've only demonstrated the simplest queries, but MongoDB has a very powerful query language that'll allow you to do more complex things. If you're not used to it, we recommend you to read the nice introduction from the official documentation. You can find this at <https://docs.mongodb.com/manual/crud>.

Nesting documents

At the beginning of this chapter, we mentioned that document-based databases, contrary to relational databases, aim to store all the data related to an entity in a single document. In our current example, if we wish to store the comments along with the post, we simply have to add a list containing information regarding each comment.

In this section, we'll implement this behavior. You should see that the functioning of MongoDB makes it very easy and straightforward.

We'll start by adding a new `comments` attribute on our `PostDB` model. You can view this in the following example:

models.py

```
class PostDB(PostBase):
    comments: List[CommentDB] = Field(default_factory=list)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb_relationship/models.py

This field is simply a list of CommentDB. We won't go into the details of the comment models, since they are quite straightforward. Notice here that we use the `list` function as the default factory for this attribute. This instantiates an empty list by default when we create a PostDB without setting any comments.

Now that we have our models, we can implement an endpoint to create a new comment. You can view it in the following example:

app.py

```
@app.post(
    "/posts/{id}/comments", response_model=PostDB, status_
    code=status.HTTP_201_CREATED
)
async def create_comment(
    comment: CommentCreate,
    post: PostDB = Depends(get_post_or_404),
    database: AsyncIOMotorDatabase = Depends(get_database),
) -> PostDB:
    await database["posts"].update_one(
        {"_id": post.id}, {"$push": {"comments": comment.
        dict()}}
    )

    post_db = await get_post_or_404(post.id, database)

    return post_db
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb_relationship/app.py

This one is slightly different from what we've seen so far. Indeed, instead of making comments a "first-class" resource with their own paths, such as for relational databases, here, we chose to nest it under the path of a single post. The motivation behind this is that, since those comments are designed to be nested under posts, it doesn't really make sense to consider them as single entities that you can work with independently.

Since we have the post ID in the path parameter, you can reuse our `get_post_or_404` dependency to retrieve the post.

Then, we trigger an `update_one` query; this time, using the `$push` operation. This is a useful operator for adding elements to a list attribute. Operators to remove elements from a list are also available. You can find a description of every update operator in the official documentation at <https://docs.mongodb.com/manual/reference/operator/update/>.

And that's it! In fact, we don't even have to modify the rest of our code. Because the comments are included in the whole document, we'll always retrieve them when querying for a post in the database. Besides, our `PostDB` model now expects a `comments` attribute, so Pydantic will take care of serializing them automatically.

That concludes this part regarding MongoDB. You've seen that its integration into a FastAPI application is very quick, especially because of its very flexible schema.

Summary

Congratulations! You've reached another big milestone in your mastering of building a REST API with FastAPI. As you know, databases are an essential part of every system; they allow you to save data in a structured way and retrieve it precisely and reliably thanks to powerful query languages. You are now able to leverage their power in FastAPI, whether they are relational databases or document-oriented databases. Additionally, you've seen the differences between working with and without an ORM to manage relational databases, and you have also learned about the importance of a good migration system when working with such databases.

Serious things can now happen: users can send and retrieve data to and from your system. However, this poses a new challenge to tackle. This data needs to be protected so that it can remain private and secure. This is exactly what we'll discuss in our next chapter: how to authenticate users and set up FastAPI for maximum security.

7

Managing Authentication and Security in FastAPI

Most of the time, you don't want everyone on the internet to have access to your API, without any restrictions on the data they can create or read. That's why you'll need to at least protect your application with a private token or have a proper authentication system to manage rights per user. In this chapter, we'll see that FastAPI provides security dependencies to help us retrieve credentials following different standards that are directly integrated into the automatic documentation. We'll also build a basic user registration and authentication system to secure our API endpoints.

Finally, we'll cover security challenges you must tackle when you want to call your API from a web application in a browser – in particular, CORS and CSRF attacks.

In this chapter, we're going to cover the following main topics:

- Security dependencies in FastAPI
- Retrieving a user and generating an access token
- Securing API endpoints for authenticated users
- Securing endpoints with access tokens
- Configuring CORS and protecting against CSRF attacks

Technical requirements

For this chapter, you'll need the Python virtual environment that we set up in *Chapter 1, Python Development Environment Setup*.

You can find all the code examples for this chapter in this book's dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter7>.

Security dependencies in FastAPI

To protect a REST API and, more generally, HTTP endpoints, lots of standards have been proposed. Here is a non-exhaustive list of the most common ones:

- **Basic HTTP authentication:** In this scheme, user credentials (usually, an identifier such as an email address and password) are put into an HTTP header called `Authorization`. The value consists of the `Basic` keyword, followed by the user credentials encoded in `Base64`. This is a very simple scheme to implement but not very secure since the password appears in every request.
- **Cookies:** Cookies are a useful way to store static data on the client side, usually on web browsers, that is sent in each request to the server. Typically, a cookie can contain a session token that can be verified by the server and linked to a specific user.
- **Tokens in the `Authorization` header:** Probably the most used header in a REST API context, this simply consists of sending a token in an HTTP `Authorization` header. The token is often prefixed by a method keyword, such as `Bearer`. On the server side, this token can be verified and linked to a specific user.

Each standard has its pros and cons and is suitable for a specific use case.

As you already know, FastAPI is mainly about dependency injection and callables that are automatically detected and called at runtime. Authentication methods are no exception: FastAPI provides most of them out of the box as security dependencies.

First, let's learn how to retrieve an access token in an arbitrary header. For this, we can use the `ApiKeyHeader` dependency, as shown in the following example:

chapter7_api_key_header.py

```
from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.params import Depends
from fastapi.security import APIKeyHeader

API_TOKEN = "SECRET_API_TOKEN"

app = FastAPI()
api_key_header = APIKeyHeader(name="Token")

@app.get("/protected-route")
async def protected_route(token: str = Depends(api_key_header)):
    if token != API_TOKEN:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN)
    return {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/chapter7_api_key_header.py

In this simple example, we hardcoded a token, `API_TOKEN`, and checked if the one that was passed in the header is equal to this token, before authorizing the endpoint to be called. To do this, we used the `APIKeyHeader` security dependency, which is designed to retrieve a value from a header. It's a class dependency that can be instantiated with arguments. It also accepts the `name` argument, which will be the name of the header it'll look for.

Then, in our endpoint, we injected this dependency to get the token's value. If it's equal to our token constant, we proceed with the endpoint logic. Otherwise, we raise a 403 error.

Our example from the *Path, router, and global dependencies* section of *Chapter 5, Dependency Injections in FastAPI*, is not very different from this one. We are simply retrieving a value from an arbitrary header and making an equality check. So, why bother with a dedicated dependency? There are two reasons:

- First, the logic to check if the header exists and retrieve its value is included in `APIKeyHeader`. When you reach the endpoint, you are sure that a token value was retrieved; otherwise, a 403 error will be thrown.
- The second, and probably most important thing, is that it's detected by the OpenAPI schema and included in its interactive documentation. This means that endpoints, including this dependency will display a lock icon, showing that it's a protected endpoint. Furthermore, you'll have access to an interface to input your token, as shown in the following screenshot. The token will then be automatically included in the requests you are making from the documentation:

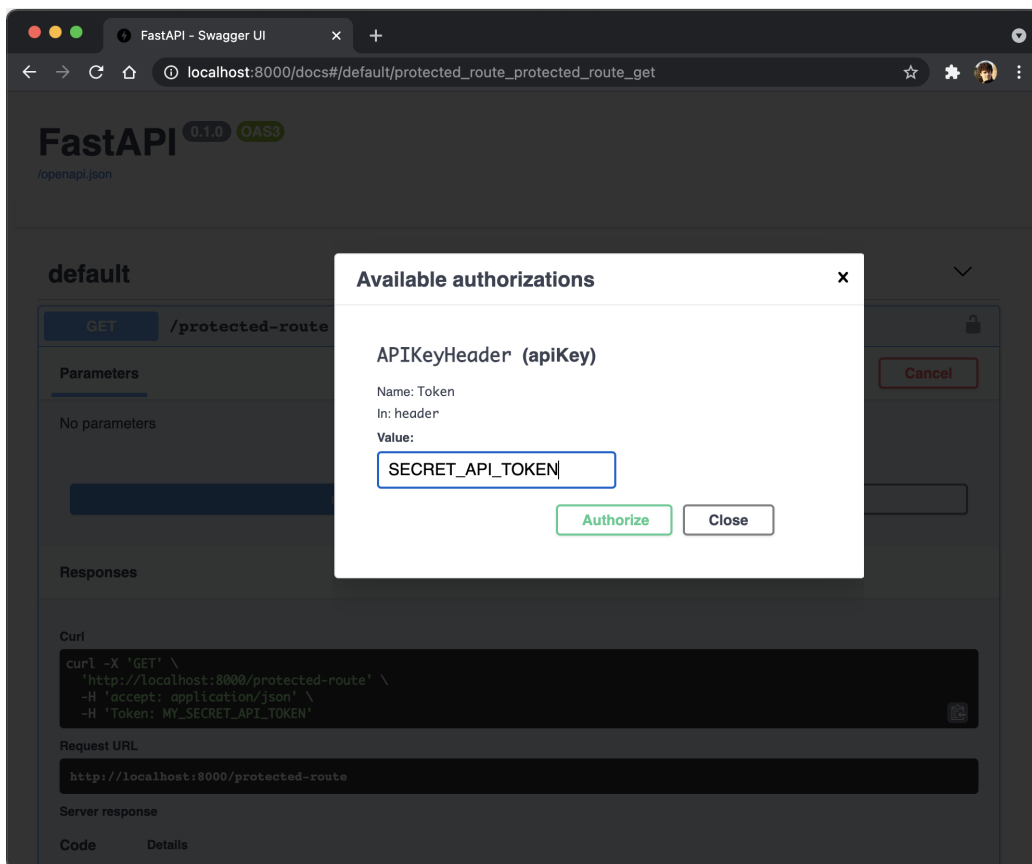


Figure 7.1 – Token authorization in interactive documentation

Of course, you can wrap the logic that checks the token value in its own dependency to reuse it across your endpoints, as shown in the following example:

chapter7_api_key_header_dependency.py

```
async def api_token(token: str =
Depends(APIKeyHeader(name="Token"))):
    if token != API_TOKEN:
        raise HTTPException(status_code=status.HTTP_403_
FORBIDDEN)

@app.get("/protected-route", dependencies=[Depends(api_token)])
async def protected_route():
    return {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/chapter7_api_key_header_dependency.py

Remember that these kinds of dependencies are very good candidates to be used as router or global dependencies to protect whole sets of routes, as we saw in *Chapter 5, Dependency Injections in FastAPI*.

This is a very basic example of adding authorization to your API. In this example, we don't have any user management; we are only checking that a token corresponds to a constant value. While it could be useful for private microservices that are not intended to be called by end users, don't consider this approach as a very secure one. First, make sure your API is always served using HTTPS to ensure your token is not exposed in the headers.

Then, if it's a private microservice, you should also consider not exposing it publicly on the internet and making sure only trusted servers can call it. Since you don't need users to make requests to this service, it's much safer than a simple token key that could be stolen.

Of course, most of the time, you'll want to authenticate real users with their own individual access token so that they can access their own data. You have probably already used a service that implements this very typical pattern:

- First, you must register an account on this service, usually by providing your email address and a password.
- Next, you can log into the service using the same email address and password. The service checks if the email address exists and that the password is valid.
- In exchange, the service provides you with a session token that can be used on subsequent requests to authenticate yourself. This way, you don't have to provide your email address and password on each request, which would be annoying and dangerous. Usually, such session tokens have a limited lifetime, which means you'll have to log in again after some time. This mitigates any security risks if the session token is stolen.

In the next section, you'll learn how to implement such a system.

Storing a user and their password securely in a database

Storing a user entity in a database is no different from storing any other entity, and you can implement this in the same way as we saw in *Chapter 6, Databases and Asynchronous ORMs*. The only thing you must be extremely cautious about is password storage. You must not store the password as plain text in your database. Why? If, unfortunately, a malicious person manages to get into your database, they'll be able to get the passwords of all your users. Since many people use the same password several times, the security of their accounts on other applications and websites would be seriously compromised.

To avoid a disaster like this, we can apply **cryptographic hash functions** to the password. The goal of those functions is to transform the password string into a hash value. They are designed to make it near impossible to retrieve the original data from the hash. Hence, even if your database is compromised, the passwords are still safe.

When users try to log in, we simply compute the hash of the password they input and compare it with the hash we have in our database. If they match, this means it's the right password.

Now, let's learn how to implement such a system with FastAPI and Tortoise ORM.

Creating models and tables

The first thing we must do is create the Pydantic models, as shown in the following example:

models.py

```
class UserBase(BaseModel):
    email: EmailStr

class Config:
    orm_mode = True

class UserCreate(UserBase):
    password: str

class User(UserBase):
    id: int

class UserDB(User):
    hashed_password: str
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/models.py>

To keep this example simple, we're only considering the email address and password in our user model. As you can see, there is a major difference between `UserCreate` and `UserDB`: the former accepts the plain text password we'll hash during registration, while the second will only keep the hashed password in the database.

Now, we can define the corresponding Tortoise model, as shown in the following example:

models.py

```
class UserTortoise(Model):
    id = fields.IntField(pk=True, generated=True)
    email = fields.CharField(index=True, unique=True,
                             null=False, max_length=255)
    hashed_password = fields.CharField(null=False,
```

```
max_length=255)
```

```
class Meta:
    table = "users"
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/models.py
```

Note that we added a unique constraint to the email column to ensure we can't have duplicate emails in our database.

Hashing passwords

Before we look at the registration endpoint, let's implement some important utility functions for hashing passwords. Fortunately, libraries exist that provide the most secure and efficient algorithms for this task. Here, we'll use `passlib`. You can install it with its optional `bcrypt` dependency, which is one of the safest hash functions at the time of writing:

```
$ pip install 'passlib[bcrypt]'
```

Now, we'll just instantiate the `passlib` classes and wrap some of their functions to make our lives easier:

password.py

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"],
                           deprecated="auto")

def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/password.py
```

`CryptContext` is a very useful class since it allows us to work with different hash algorithms. If, one day, a better algorithm than `bcrypt` emerges, we can just add it to our allowed schemes. New passwords will be hashed using the new algorithm, but existing passwords will still be recognized (and optionally upgraded to the new algorithm).

Implementing registration routes

Now, we have all the elements to create a proper registration route. Once again, it'll be very similar to what we saw earlier. The only thing we must remember is to hash the password before inserting it into our database.

Let's look at the implementation:

app.py

```
@app.post("/register", status_code=status.HTTP_201_CREATED)
async def register(user: UserCreate) -> User:
    hashed_password = get_password_hash(user.password)

    try:
        user_tortoise = await UserTortoise.create(
            **user.dict(), hashed_password=hashed_password
        )
    except IntegrityError:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Email already exists"
        )

    return User.from_orm(user_tortoise)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/app.py>

As you can see, we are calling `get_password_hash` on the input password before inserting the user into the database thanks to Tortoise. Note that we are catching a possible `IntegrityError` exception, which means we're trying to insert an email that already exists.

Also, notice that we took care to return the user with the `User` model, not the `UserDB` model. By doing this, we're ensuring that `hashed_password` is not part of the output. Even hashed, it's generally not advised to leak it into the API responses.

Great! We now have a proper user model and users can create a new account with our API. The next step is to allow them to log in and give them an access token.

Retrieving a user and generating an access token

After successful registration, the next step is being able to log in: the user will send their credentials and receive an authentication token to access the API. In this section, we'll implement the endpoint that allows this. Basically, we'll get the credentials from the request payload, retrieve the user with the given email, and verify their password. If the user exists and their password is valid, we'll generate an access token and return it in the response.

Implementing a database access token

First, let's think about the nature of this access token. It should be a data string that uniquely identifies a user that is impossible to forge by a malicious third party. In this example, we will take a simple but reliable approach: we'll generate a random string and store it in a dedicated table in our database, with a foreign key referring to the user.

This way, when an authenticated request arrives, we simply have to check whether it exists in the database and look for the corresponding user. The advantage of this approach is that tokens are centralized and can easily be invalidated if they are compromised; we only need to delete them from the database.

The first step is to implement the Pydantic and Tortoise models for this new entity. Let's have a look at the Pydantic model first:

models.py

```
class AccessToken(BaseModel):
    user_id: int
    access_token: str = Field(default_factory=generate_token)
```

```
    expiration_date: datetime = Field(default_factory=get_
expiration_date)
```

```
class Config:
```

```
    orm_mode = True
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/models.py>

Here, we have three fields:

- `user_id`, which will let us identify the user that corresponds to this token.
- `access_token`, the string that will be passed in the requests to authenticate them. Notice that we defined the `generate_token` function as the default factory; it's a simple function living in `password.py` that generates a random secure passphrase. Under the hood, it relies on the standard `secrets` module.
- `expiration_date`, which is the date and time when the access token won't be valid anymore. It's always a good idea to make access tokens expire to mitigate the risk if they are stolen. Here, the `get_expiration_date` factory sets a default validity of 24 hours.

Now, let's have a look at the corresponding Tortoise model:

models.py

```
class AccessTokenTortoise(Model):
    access_token = fields.CharField(pk=True,
                                    max_length=255)
    user = fields.ForeignKeyField("models.UserTortoise",
                                   null=False)
    expiration_date = fields.DatetimeField(null=False)

    class Meta:
        table = "access_tokens"
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/models.py>

The implementation here is quite straightforward. Notice that we chose to directly use `access_token` as a primary key.

Implementing a login endpoint

Now, let's think about the login endpoint. Its goal is to take credentials in the request payload, retrieve the corresponding user, check the password, and generate a new access token. Its implementation is quite straightforward, apart from one thing: the model that's used to handle the request. You'll see why thanks to the following example:

app.py

```
@app.post("/token")
async def create_token(
    form_data: OAuth2PasswordRequestForm =
    Depends(OAuth2PasswordRequestForm),
):
    email = form_data.username
    password = form_data.password
    user = await authenticate(email, password)

    if not user:
        raise HTTPException(status_code=status.HTTP_401_
        UNAUTHORIZED)

    token = await create_access_token(user)

    return {"access_token": token.access_token,
            "token_type": "bearer"}
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/app.py>

As you can see, we retrieve the request data thanks to the `OAuth2PasswordRequestForm` module, which is provided by FastAPI in its security module. It expects several fields, especially `username` and `password`, in a form encoding rather than JSON.

Why do we use this class? The main benefit of using this class is that it's completely integrated into the OpenAPI schema. This means that the interactive documentation will be able to automatically detect it and present a proper authentication form behind the **Authorize** button, as shown in the following screenshot:

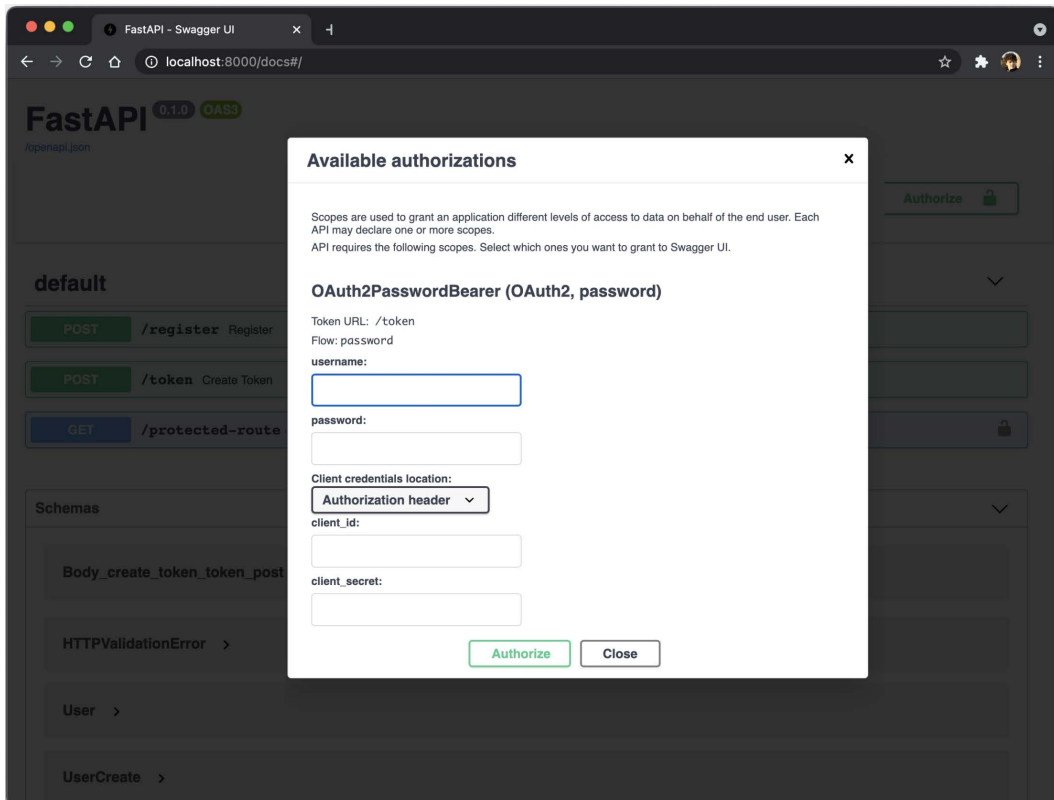


Figure 7.2 – OAuth2 authorization in interactive documentation

But that's not all: it will be able to automatically retrieve the returned access token and set the proper authorization header in subsequent requests. The authentication process is handled transparently by the interactive documentation.

This class follows the OAuth2 protocol, which means you also have fields for the client ID and secret. We won't learn how to implement the complete OAuth2 protocol here, but note that FastAPI provides all the tools needed to do so properly. For our project, we'll just stick with a username and a password. Notice that, following the protocol, the field is named *username*, regardless of whether we are using an email address to identify the user. This isn't a big deal; we just have to remember it while retrieving it.

The rest of the path operation function is quite simple: first, we try to retrieve a user from this email and password. If no corresponding user is found, we raise a 401 error. Otherwise, we generate a new access token before returning it. Notice that the response structure also includes the `token_type` property. This allows the interactive documentation to automatically sets the authorization headers.

In the following example, we'll look at the implementation of the `authenticate` and `create_access_token` functions. We won't go into too much detail here as they are quite simple:

authentication.py

```
async def authenticate(email: str, password: str) ->
Optional[UserDB]:
    try:
        user = await UserTortoise.get(email=email)
    except DoesNotExist:
        return None

    if not verify_password(password, user.hashed_password):
        return None

    return UserDB.from_orm(user)

async def create_access_token(user: UserDB) -> AccessToken:
    access_token = AccessToken(user_id=user.id)
    access_token_tortoise = await AccessTokenTortoise.
create(**access_token.dict())

    return AccessToken.from_orm(access_token_tortoise)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/authentication.py>

Notice that we defined a function called `verify_password` to check the validity of the password. Once again, it uses `passlib` under the hood, which takes care of comparing the hashes of the passwords.

Password Hash Upgrade

To keep this example simple, we implemented a simple password comparison. Usually, it's good practice to implement a mechanism to upgrade the password hash at this stage. Imagine that a new and more robust hash algorithm has been introduced. We can take this opportunity to hash the password with this new algorithm and store it in a database. `passlib` includes a function for verifying and upgrading the hash with one operation. You can learn more about this in the following documentation: <https://passlib.readthedocs.io/en/stable/narr/context-tutorial.html#integrating-hash-migration>.

We've almost achieved our goal! Users can now log in and get a new access token. All we need to do now is implement a dependency to retrieve the `Authorization` header and verify this token!

Securing endpoints with access tokens

Previously, we learned how to implement a simple dependency to protect an endpoint with a header. Here, we'll also retrieve a token from a request header, but then, we'll have to check the database to see if it's valid. If it is, we'll be able to return the corresponding user.

Let's see what our dependency looks like:

app.py

```
async def get_current_user(
    token: str = Depends(OAuth2PasswordBearer(tokenUrl="/
token")),
) -> UserTortoise:
    try:
        access_token: AccessTokenTortoise = await
AccessTokenTortoise.get(
```

```
        access_token=token, expiration_date__gte=datetime.datetime.now()
    ).prefetch_related("user")
    return cast(UserTortoise, access_token.user)
except DoesNotExist:
    raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/app.py>

The first thing to notice is that we used the `OAuth2PasswordBearer` dependency from FastAPI. It goes hand in hand with `OAuth2PasswordRequestForm`, which we saw in the previous section. It not only checks for the access token in the `Authorization` header, but it also informs the OpenAPI schema that the endpoint to get a fresh token is `/token`. This is the purpose of the `tokenUrl` argument. This is how the automatic documentation can automatically call the access token endpoint in the login form we saw earlier.

Then we performed a database query with Tortoise. We applied two clauses: one to match the token we got and another to ensure that the expiration date is in the future. The `__gte` syntax is a **filter modifier**: it allows us to specify the comparison operator to apply when comparing values. Here, `gte` means "greater than or equal to." You can find a list of every filter that's available in Tortoise in the official documentation: <https://tortoise-orm.readthedocs.io/en/latest/query.html#filtering>. Notice that we also prefetched the related user so that we can directly return it. However, if no corresponding record is found in the database, we raise a 401 error.

And that's it! Our whole authentication system is complete. Now, we can protect our endpoints simply by injecting this dependency. We even have access to the user data so that we can tailor the response according to the current user. You can see this in the following example:

app.py

```
@app.get("/protected-route", response_model=User)
async def protected_route(user: UserDB = Depends(get_current_user)):
    return User.from_orm(user)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/authentication/app.py>

With that, you've learned how to implement a whole registration and authentication system from scratch. We voluntarily kept it simple to focus on the most important points, but it's a good base you can expand.

The patterns we showed here are good candidates for a REST API, which is called externally by other client programs. However, you may wish to call your API from a very common piece of software: the browser. In this case, there are some additional security considerations to take care of.

Configuring CORS and protecting against CSRF attacks

Nowadays, lot of software are designed to be used in a browser through an interface built with HTML, CSS, and JavaScript. Traditionally, web servers were responsible for handling browser requests and returning an HTML response, ready to be shown. This is a common use case for frameworks such as Django.

For a few years now, there has been a shift in that pattern. With the emergence of JavaScript frameworks, such as Angular, React, and Vue, we tend to have a clear separation between the frontend, a highly interactive user interface powered by JavaScript, and the backend. Thus, those backends are now only responsible for data storage and retrieving and executing business logic. This is a task that REST APIs are very good at! From the JavaScript code, the user interface can then just spawn requests to your API and handle the result to present it.

However, we must still handle authentication: we want our user to be able to log in on the frontend application and be able to make authenticated requests to the API. While an `Authorization` header, as we've seen so far, could work, there is a better way to handle authentication when working in browsers: **cookies!**

Cookies are designed to store user information in browser memory and are sent automatically in every request made to your server. They have been supported for years, and browsers integrate lots of mechanisms to make them safe and reliable.

However, this comes with some security challenges. Websites are very common targets for hackers and lots of attacks have emerged over the years.

One of the most typical is **Cross-Site Request Forgery (CSRF)**. In this scenario, an attacker on another website tries to trick a user who is currently authenticated with your application to perform a request on your server. Since browsers tend to send cookies with every request, your server wouldn't be able to tell that the request was actually forged. Since it's the users themselves who unintentionally launched the malicious request, these kinds of attacks don't aim to steal data but execute operations that change the state of the application, such as changing an email address or making a money transfer.

Obviously, we should be prepared for these kinds of risks and have measures in place to mitigate them.

Understanding CORS and configuring it in FastAPI

When you have a clearly separated frontend application and a REST API backend, they typically are not served from the same sub-domain. For example, the frontend may be available from `www.myapplication.com`, while the REST API may be available from `api.myapplication.com`. As we mentioned in the introduction, we would like to make requests to this API from our frontend application, in JavaScript.

However, browsers don't allow **cross-origin HTTP requests**, meaning domain A can't make requests to domain B. This follows what is called a **same-origin policy**. This is a good thing in general as it's the first barrier to preventing CSRF attacks.

To experience this behavior, we'll run a simple example. In our example repository, the `chapter7/cors` folder contains a FastAPI app called `app_without_cors.py` and a simple HTML file called `index.html` that contains some JavaScript for performing HTTP requests.

First, let's run the FastAPI application using the usual `uvicorn` command:

```
$ uvicorn chapter7.cors.app_without_cors:app
```

This will launch the FastAPI application on port 8000 by default. On another terminal, we'll serve the HTML file using the built-in Python HTTP server. It's a simple server, but it's ideal for quickly serving static files. We can launch it on port 9000 thanks to the following command:

```
$ python -m http.server --directory chapter7/cors 9000
```

Starting Several Terminals

On Linux and macOS, you should be able to simply start a new terminal by creating a new window or tab. On Windows and WSL, you can also have several tabs if you're using the Windows terminal application: <https://www.microsoft.com/en-us/p/windows-terminal/9n0dx20hk701?activetab=pivot:overviewtab>.

Otherwise, you can simply click on the Ubuntu shortcut in your **Start** menu to start another terminal.

We now have two running servers – one on `localhost:8000` and one on `localhost:9000`. Strictly speaking, since they are on different ports, they are of different origins; so, it's a good setup to try out cross-origin HTTP requests.

In your browser, go to `http://localhost:9000`. You'll see the simple application implemented in `index.html`, as shown in the following screenshot:

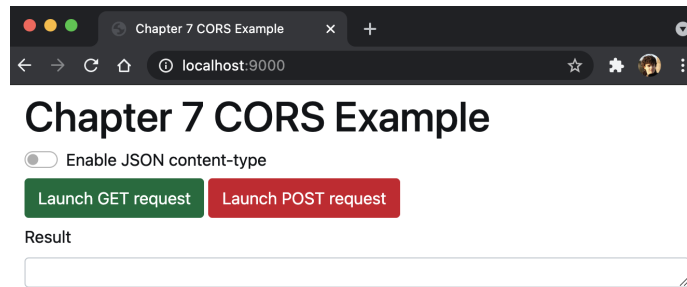


Figure 7.3 – Simple application for trying out CORS policies

There are two buttons that initiate GET and POST requests to our FastAPI application on port 8000. If you click on either of those, you'll have a message in the error area stating **Failed to fetch**. If you look at the browser console in the development tools section, you'll see that the request has failed because there isn't a CORS policy, as shown in the following screenshot. That's what we wanted – by default, browsers block cross-origin HTTP requests:

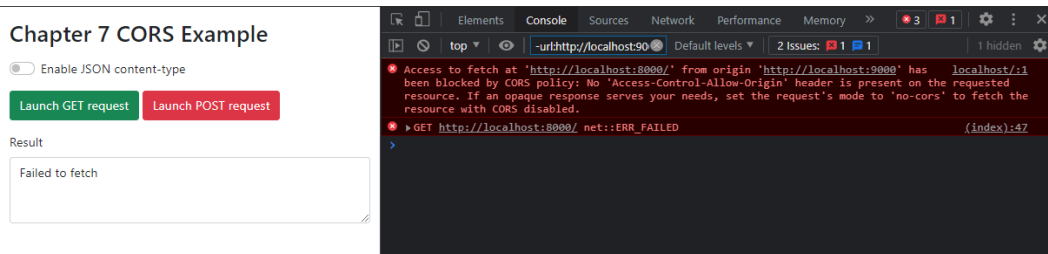
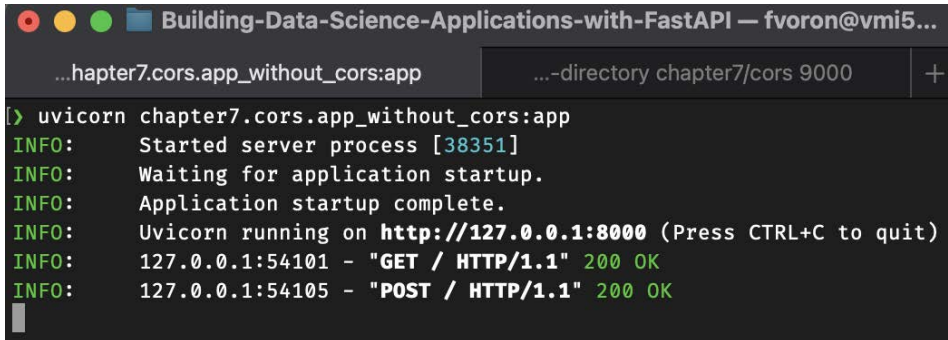


Figure 7.4 – CORS error in a browser console

However, if you look at the terminal running the FastAPI application, you'll see an output similar to the following:



```

Building-Data-Science-Applications-with-FastAPI — fvoron@vmi5...
...hapter7.cors.app_without_cors:app  ...-directory chapter7/cors 9000 +
> uvicorn chapter7.cors.app_without_cors:app
INFO: Started server process [38351]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: 127.0.0.1:54101 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:54105 - "POST / HTTP/1.1" 200 OK

```

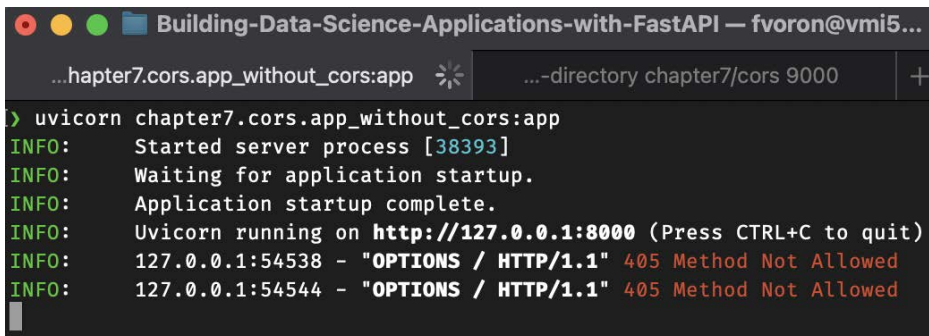
Figure 7.5 – Uvicorn output when performing simple requests

Clearly, both the `GET` and `POST` requests have been received and processed: we even returned a `200` status. So, what does this mean? In this case, the browser does send the request to the server. The lack of a CORS policy only forbids it to read the response; the request is still executed.

It happens for requests that the browser considers as simple requests. Simply put, simple requests are the ones using the methods `GET`, `POST` or `HEAD` that don't set custom headers or unusual content types. You can learn more about simple requests and their conditions by going to the following MDN page about CORS: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#simple_requests.

This means that, for simple requests, the same-origin policy is not enough to protect us against CSRF attacks.

You may have noticed that our simple web application has a toggle to **Enable JSON content-type**. Enable it and perform the `GET` and `POST` requests again. On your FastAPI terminal, you should have an output similar to the following:



```

Building-Data-Science-Applications-with-FastAPI — fvoron@vmi5...
...hapter7.cors.app_without_cors:app  ...-directory chapter7/cors 9000 +
> uvicorn chapter7.cors.app_without_cors:app
INFO: Started server process [38393]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: 127.0.0.1:54538 - "OPTIONS / HTTP/1.1" 405 Method Not Allowed
INFO: 127.0.0.1:54544 - "OPTIONS / HTTP/1.1" 405 Method Not Allowed

```

Figure 7.6 – Uvicorn output when receiving preflight requests

As you can see, our server received two strange requests with the `OPTIONS` method. This is what we call **preflight requests** in the context of CORS policies. Those requests are initiated by the browser before it performs the actual request when it doesn't consider it a "simple request." Here, we added the `Content-Type` header with a value of `application/json`, which is against the conditions of simple requests.

By performing this preflight request, the browser expects the server to provide information about what it is and isn't allowed to do in terms of cross-origin HTTP requests. Since we've not implemented anything here, our server can't provide a response to this preflight request. Hence, the browser stops there and doesn't proceed with the actual request.

And that's basically CORS: the server answers preflight queries with a set of HTTP headers that provide information to the browser about whether it's allowed to make the request or not. In that sense, CORS doesn't make your application more secure, it's quite the contrary: it allows to relax some rules so that a frontend application can make requests to a backend living on another domain. That's why it's crucial to configure them properly, so that they don't expose you to dangerous attacks.

Fortunately, it's fairly easy to do this with FastAPI. All we need to do is import and add the `CORSMiddleware` class provided by Starlette. You can see what it looks like in the following example:

app_with_cors.py

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:9000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
    max_age=-1, # Only for the sake of the example.
    # Remove this in your own project.
)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/cors/app_with_cors.py

A middleware is a special class that adds global logic to an ASGI application performing things before the request is handled by your path operation functions, and also after to possibly alter the response. FastAPI provides the `add_middleware` method for wiring such middleware into your application.

Here, `CORSMiddleware` will catch preflight requests sent by the browser and return the appropriate response with the CORS headers corresponding to your configuration. You can see that there are options to finely tune the CORS policy to your needs.

The most important one is probably `allow_origins`, which is the list of origins allowed to make requests to your API. Since our HTML application is served from `http://localhost:9000`, this is what we put here in this argument. If the browser tries to make requests from any other origin, it will stop as it's not authorized to do so by CORS headers.

The other interesting argument is `allow_credentials`. By default, browsers don't send cookies for cross-origin HTTP requests. If we wish to make authenticated requests to our API, we need to allow this via this option.

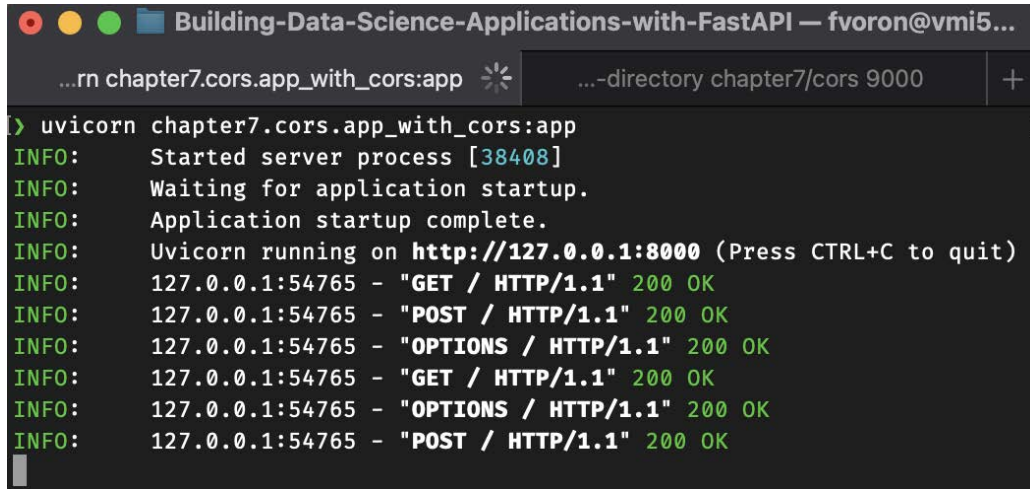
We can also finely tune the allowed methods and headers that are sent in the request. You can find a complete list of arguments for this middleware in the official Starlette documentation: <https://www.starlette.io/middleware/#corSMiddleware>.

Let's quickly talk about the `max_age` parameter. This parameter allows you to control the cache duration of the CORS responses. Having to perform a preflight request before the actual one is an expensive operation. To improve performance, browsers can cache the response so that they don't have to do this every time. Here, we are disabling caching with a value of `-1` to make sure you see the behavior of the browser in this example. In your projects, you can remove this argument so that you have a proper cache value.

Now, let's see how our web application behaves with this CORS-enabled application. Stop the previous FastAPI app and run this one using the usual command:

```
$ uvicorn chapter7.cors.app_with_cors:app
```

Now, if you try to perform the requests from the HTML application, you should see a working response in each case, both with and without a JSON content type. If you look at the FastAPI terminal, you should see an output similar to the following:



```

> uvicorn chapter7.cors.app_with_cors:app
INFO: Started server process [38408]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: 127.0.0.1:54765 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:54765 - "POST / HTTP/1.1" 200 OK
INFO: 127.0.0.1:54765 - "OPTIONS / HTTP/1.1" 200 OK
INFO: 127.0.0.1:54765 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:54765 - "OPTIONS / HTTP/1.1" 200 OK
INFO: 127.0.0.1:54765 - "POST / HTTP/1.1" 200 OK

```

Figure 7.7 – Uvicorn output with CORS headers

The two first requests are the "simple requests," which don't need a preflight request according to the browser rules. Then, we can see the requests that were performed with the JSON content type enabled. Before the GET and POST requests, an OPTIONS request was performed: the preflight request!

Thanks to this configuration, you can now make cross-origin HTTP requests between your frontend application and your backend living on another origin. Once again, it's not something that'll improve the security of your application, but it allows you to make this specific scenario work while keeping it secure from the rest of the web.

Even if those policies can be a first layer of defense against CSRF, this doesn't mitigate the risk completely. Indeed, the "simple requests" are still an issue: POST requests are allowed and, even if the response cannot be read, it's actually executed on the server.

Now, let's learn how to implement a pattern so that we're completely safe from such attacks: the **double-submit cookie**.

Implementing double-submit cookies to prevent CSRF attacks

As we mentioned previously, when relying on cookies to store user credentials, we are exposed to CSRF attacks since browsers will automatically send the cookies to your server. This is especially true for what the browser considers "simple requests", which don't enforce the CORS policy before the request is executed. There are also other attack vectors involving traditional HTML form submissions or even the `src` attribute of the image tag.

For all these reasons, we need to have another layer of security to mitigate this risk. Once again, this is only necessary if you plan to use your API from a browser application and use cookies for authentication.

To help you understand this, we've built a new example application that uses a cookie to store the user access token. It's very similar to the one we saw at the beginning of this chapter; we only modified it so that it looks for the access token in a cookie rather than in a header.

To make this example work, you'll have to install the `starlette-csrf` library. We'll explain what it does a bit later in this section. For now, just run the following command:

```
$ pip install starlette-csrf
```

In the following example, you can see the login endpoint that sets a cookie with the access token value:

app.py

```
@app.post("/login")
async def login(response: Response, email: str = Form(...),
password: str = Form(...)):
    user = await authenticate(email, password)

    if not user:
        raise HTTPException(status_code=status.HTTP_401_
UNAUTHORIZED)

    token = await create_access_token(user)

    response.set_cookie(
        TOKEN_COOKIE_NAME,
        token.access_token,
        max_age=token.max_age(),
        secure=True,
        httponly=True,
        samesite="lax"
    )
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/csrf/app.py>

Notice that we used the `Secure` and `HttpOnly` flags for the resulting cookie. This ensures that it's sent only through HTTPS connection and that its value can't be read from JavaScript, respectively. While this is not enough to prevent every kind of attack, it's crucial for such sensitive information.

Besides, we also set the `SameSite` flag to `lax`. It's a quite recent flag that allows us to control how the cookie is sent in a cross-origin context. `lax` is the default value in most browsers and allows the cookie to be sent to sub-domains of the cookie domain but prevent it for other sites. In a sense, it's designed to be the built-in and standard protection against CSRF. However, other CSRF mitigation techniques, like the one we'll implement here, are still needed currently. Indeed, older browsers that are not compatible with the `SameSite` flag are still vulnerable.

Now, when checking for the authenticated user, we'll just have to retrieve the token from the cookie that was sent in the request. Once again, FastAPI provides a security dependency to help with this called `APIKeyCookie`. You can see it in the following example:

app.py

```
async def get_current_user(
    token: str = Depends(APIKeyCookie(name=TOKEN_COOKIE_NAME)),
) -> UserTortoise:
    try:
        access_token: AccessTokenTortoise = await
        AccessTokenTortoise.get(
            access_token=token, expiration_date_gte=datetime.datetime.now()
        ).prefetch_related("user")
        return cast(UserTortoise, access_token.user)
    except DoesNotExist:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/csrf/app.py>

And that's basically it! The rest of the code remains the same. Now, let's implement an endpoint that allows us to update the email address of the authenticated user. You can see this in the following example:

app.py

```
@app.post("/me", response_model=User)
async def update_me(
    user_update: UserUpdate, user: UserTortoise = Depends(get_current_user)
):
    user.update_from_dict(user_update.dict(exclude_unset=True))
    await user.save()

    return User.from_orm(user)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/csrf/app.py>

The implementation is not very surprising and follows what we've seen so far. However, it exposes us to a CSRF threat. As you can see, it uses the POST method. If we make a request in the browser to this endpoint, without any special header, it will consider it as a simple request and execute it. Therefore, an attacker could change the email of a currently authenticated user, which is a major threat.

This is exactly why we need CSRF protection here. In the context of a REST API, the most straightforward technique is the double submit cookie pattern. Here is how it works:

1. The user makes a first request with a method that's considered safe. Typically, this is a GET request.
2. In response, it receives a cookie containing a secret random value; that is, the CSRF token.
3. When making an unsafe request, such as POST, the user will read the CSRF token in the cookies and put the exact same value in a header. Since the browser also sends the cookies it has in memory, the request will contain the token both in the cookie and the header. That's why it's called **double submit**.
4. Before processing the request, the server will compare the CSRF token provided in the header with the one present in the cookie. If they match, it can process the request. Otherwise, it'll throw an error.

This is safe for two reasons:

- An attacker on a third-party website can't read the cookies for a domain they don't own. Thus, they have no way of retrieving the CSRF token value.
- Adding a custom header is against the conditions of "simple requests". Hence, the browser will have to make a preflight request before sending the request, enforcing the CORS policy.

This is a widely used pattern that works well to prevent such risks. This is why we installed `starlette-csrf` at the beginning of this section: it provides a piece of middleware for implementing it.

We can use it just like any other middleware, as shown in the following example:

app.py

```
app.add_middleware(  
    CSRFMiddleware,  
    secret=CSRF_TOKEN_SECRET,  
    sensitive_cookies={TOKEN_COOKIE_NAME},  
    cookie_domain="localhost",  
)
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter7/csrf/app.py>

We set several important arguments here. First, we have the secret, which should be a strong passphrase that's used to sign the CSRF token. Then, we have `sensitive_cookies`, which is a set of cookie names that should trigger the CSRF protection. If no cookie is present or if the provided ones are not critical, we can bypass the CSRF check. It's also useful if you have other authentication methods available that don't rely on cookies, such as Authorization headers, that are not vulnerable to CSRF. Finally, setting a cookie domain will allow you to retrieve the cookie containing the CSRF token, even if you are on a different subdomain; this is necessary in a cross-origin situation.

That's all you need to have the necessary protection ready. To ease the process of getting a fresh CSRF token, we implemented a minimal GET endpoint called `/csrf`. Its sole purpose is to provide us with a simple way to set the CSRF token cookie. We can call it directly when we load our frontend application.

Now, let's try it out in our situation. As we did in the previous section, we'll run the FastAPI application and the simple HTML application on two different ports. To do this, just run the following commands:

```
$ uvicorn chapter7.csrf.app:app
```

This will run the FastAPI application on port 8000. Now, run the following command:

```
$ python -m http.server --directory chapter7/csrf 9000
```

The frontend application is now live on `http://localhost:9000`. Open it in your browser. You should see an interface similar to the following:

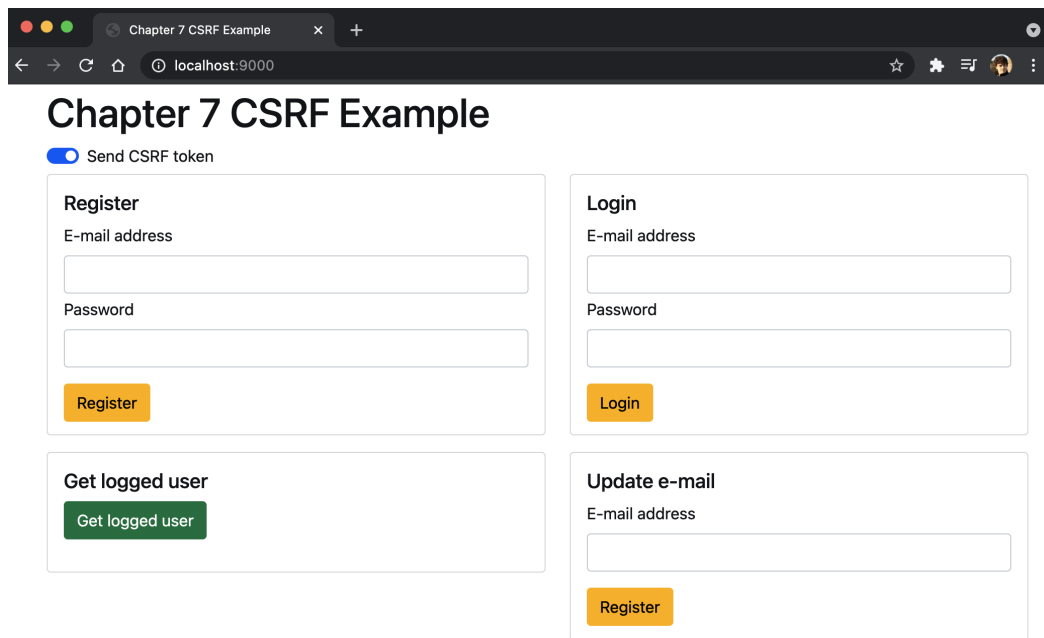


Figure 7.8 – Simple application to try out the CSRF protected API

Here, we've added forms to interact with our API endpoints: register, login, get authenticated user, and update them. If you try them out, they should work without any issue. If you have a look at the requests that were sent in the network tab of the development tools section, you'll see that the CSRF token is present in the cookies and in a header called `x-csrf-token`.

At the top, there is a toggle to prevent the application from sending the CSRF token in the header. If you disable it, you'll see that all `POST` operations will result in an error.

Great! We are now safe from CSRF attacks! Most of the work here is done by the middleware, but it's interesting to understand how it works under the hood and how it protects your application. Bear in mind, however, that it comes with a drawback: it will break the interactive documentation. Indeed, it's not designed to retrieve the CSRF token from the cookie and put it in the headers in each request. Unless you plan of authenticating in another way (through a token in a header, for example), you won't be able to directly call your endpoints in the documentation.

Summary

That's all for this chapter, which covered authentication and security in FastAPI. We saw that implementing a basic authentication system is quite easy thanks to the tools provided by FastAPI. We've shown you one way to do this, but there are plenty of other good patterns out there to tackle this challenge. However, when working on this matter, always keep security in mind and be sure that you don't expose your application and your users' data to dangerous threats. In particular, you've seen that CSRF attacks have to be taken care of when designing a REST API that will be used in a browser application. A good source to understand all the security risks involved in a web application is the OWASP Cheat Sheet Series: <https://cheatsheetseries.owasp.org>.

With that, we've covered most of the important subjects concerning FastAPI application development. In the next chapter, we'll learn how to work with a recent technology that's integrated with FastAPI that allows to have real time two-way communication between the client and the server: websockets.

8

Defining WebSockets for Two-Way Interactive Communication in FastAPI

The **HyperText Transfer Protocol (HTTP)** is a simple yet powerful technique to send or receive data to and from a server. As we've seen, the principles of request and response are at the core of this protocol: when developing our **application programming interface (API)**, our goal is to process the incoming request and build a response for the client. Thus, in order to get data from the server, the client always has to initiate a request first. In some contexts, however, this may not be very convenient. Imagine a typical chat application: when a user receives a new message, we would like them to be notified immediately by the server. Working only with HTTP, we would have to make requests every second to check if new messages have arrived, which would be a massive waste of resources. This is why a new protocol has emerged: **WebSocket**. The goal of this protocol is to open a communication channel between a client and a server so that they can exchange data in real time, in both directions.

In this chapter, we're going to cover the following main topics:

- Understanding the principles of two-way communication with WebSockets
- Creating a WebSocket with FastAPI
- Handling multiple WebSocket connections and broadcasting messages

Technical requirements

You'll need a Python virtual environment, as we set up in *Chapter 1, Python Development Environment Setup*.

For the *Handling multiple WebSocket connections and broadcasting messages* section, you'll need a running Redis server on your local computer. The easiest way is to run it as a Docker container. If you've never used Docker before, we recommend you read the *Getting started* tutorial in the official documentation at <https://docs.docker.com/get-started/>. Once done, you'll be able to run a Redis server with this simple command:

```
$ docker run -d --name fastapi-redis -p 6379:6379 redis
```

This will make it available on your local computer on port 6379.

You'll find all the code examples of this chapter in the dedicated GitHub repository at <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter8>.

Understanding the principles of two-way communication with WebSockets

You have probably noticed that the name **WebSockets** is a direct reference to the traditional concept of **sockets** in Unix systems. While technically unrelated, they achieve the same goal: to open a *communication channel between two applications*. As we said in the introduction, HTTP works only on a request-response principle, which makes the implementation of applications that need real-time communication between the client and the server difficult and inefficient.

WebSockets try to solve that by opening a full-duplex communication channel, meaning that messages can be sent in both directions and possibly at the same time. Once the channel is opened, the server can send messages to the client without having to wait for a request from the client.

Even if HTTP and WebSocket are different protocols, WebSockets have been designed to work with HTTP. Indeed, when opening a WebSocket, the connection is first initiated using an HTTP request and then upgraded to a WebSocket tunnel. This makes it compatible out of the box with traditional 80 and 443 ports, which is extremely convenient because we can easily add this feature over existing web servers without the need for an extra process.

WebSockets also share another similarity with HTTP: **Uniform Resource Identifiers (URIs)**. As with HTTP, WebSockets are identified through classic URIs, with a host, a path, and query parameters. Furthermore, we also have two schemes: **ws (WebSocket)** for unsecure connections and **wss (WebSocket Secure)** for **Secure Sockets Layer/Transport Layer Security (SSL/TLS)**-encrypted connections.

Finally, this protocol is nowadays well supported in browsers, and opening a connection with a server involves just a few lines of JavaScript, as we'll see in this chapter.

However, handling this two-way communication channel is quite different from handling traditional HTTP requests. Since things happen in real time and in both directions, we'll see that we have to think differently from what we are used to. In FastAPI, the asynchronous nature of the WebSocket implementation will greatly help us in finding our way through that.

Creating a WebSocket with FastAPI

Thanks to Starlette, FastAPI has built-in support to serve WebSockets. As we'll see, defining a WebSocket endpoint is quick and easy, and we'll be able to get started in minutes. However, things will get more complex as we try to add more features to our endpoint logic. Let's start simple, with a WebSocket that waits for messages and simply echoes them back.

In the following example, you'll see the implementation of such a simple case:

app.py

```
from fastapi import FastAPI, WebSocket
from starlette.websockets import WebSocketDisconnect

app = FastAPI()

@app.websocket("/ws")
```

```
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    try:
        while True:
            data = await websocket.receive_text()
            await websocket.send_text(f"Message text was:
{data}")
    except WebSocketDisconnect:
        await websocket.close()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/echo/app.py>

The code is quite understandable by itself, but let's focus on the important parts that differ from classic HTTP endpoints.

First of all, you see that FastAPI provides a special `websocket` decorator to create a WebSocket endpoint. As for regular endpoints, it takes as an argument the path at which it'll be available. However, other arguments not making sense in this context, such as the status code or response model, are not available.

Then, in the path operation function, we can inject a `WebSocket` object, which will provide us all the methods to work with the WebSocket, as we'll see.

The first method we are calling in the implementation is `accept`. This method should be called first as it tells the client that we agree to open the tunnel.

After that, you see that we start an infinite loop. That's the main difference with an HTTP endpoint: since we are opening a communication channel, it'll remain open until the client or the server decides to close it. While it's open, they can exchange as many messages as they need, hence the infinite loop is here to keep it open and repeat the logic until the tunnel is closed.

Inside the loop, we make a first call to the `receive_text` method. As you may have guessed, this returns us the data sent by the client in plain text format. It's important here to understand that *this method will block until data is received from the client*. Until that event, we won't proceed with the rest of the logic.

We see here the importance of asynchronous input/output, as we presented in *Chapter 2, Python Programming Specificities*. By creating an infinite loop waiting for incoming data, we could have blocked the whole server process in a traditional blocking paradigm. Here, thanks to the event loop, the process is able to answer other requests made by other clients while we are waiting for this one.

When data is received, the method returns the text data and we can proceed with the next line. Here, we simply send back the message to the client thanks to the `send_text` method. Once done, we are going back to the beginning of the loop to wait for another message.

You probably noticed that the whole loop is wrapped inside a `try . . except` statement. This is necessary to *handle client disconnection*. Indeed, our server will most of the time be blocked at the `receive_text` line, waiting for client data. If the client decides to disconnect, the tunnel will be closed and the `receive_text` call will fail, with a `WebSocketDisconnect` exception. That's why it's important to catch it to break the loop and properly call `disconnect` on the server side.

Let's try it! You can run the FastAPI application, as usual, thanks to the Uvicorn server. Here's the command you'll need:

```
$ uvicorn chapter8.echo.app:app
```

Our client will be a simple **HyperText Markup Language (HTML)** page with some JavaScript code to interact with the WebSocket. We'll quickly go through this code after the demonstration. To run it, we can simply serve it with the built-in Python server, as follows:

```
$ python -m http.server --directory chapter8/echo 9000
```

Starting several terminals

On Linux and macOS, you should be able to simply start a new terminal by creating a new window or tab. On Windows and **Windows Subsystem for Linux (WSL)**, you can also have several tabs if you use the Windows terminal application (see <https://www.microsoft.com/en-us/p/windows-terminal/9n0dx20hk701?activetab=pivot:overviewtab> for more information). Otherwise, you can simply click again on the Ubuntu shortcut in your **Start** menu to start another terminal.

This will serve our HTML page on port 9000 of your local machine. If you open the `http://localhost:9000` address, you'll see a simple interface like the one shown here:

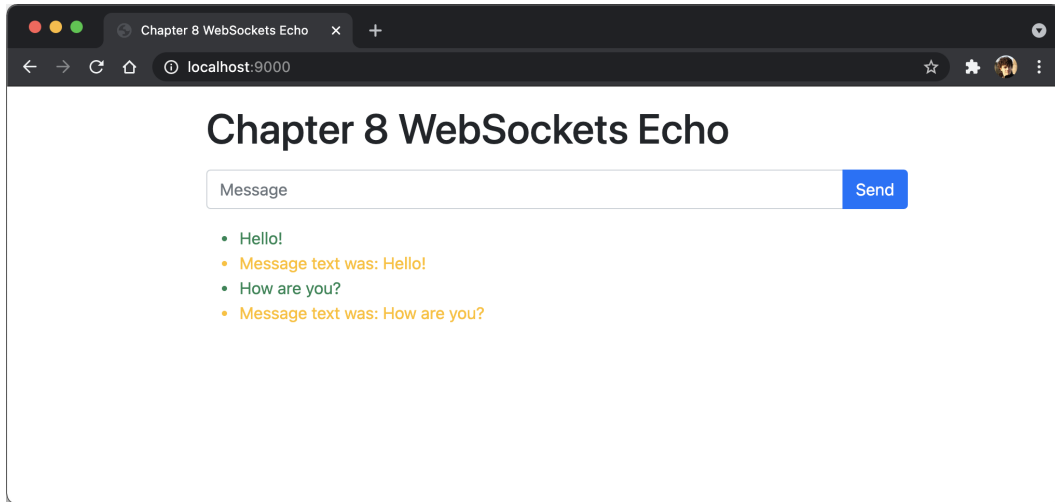


Figure 8.1 – Simple application to try WebSocket

You have a simple input form, allowing you to send messages to the server through the WebSocket. They appear in green in the list below. The server echoes back your messages, which then appear in yellow in the list.

You can see what's happening under the hood by opening the **Network** tab in the developer tools of your browser. Reload the page to force the WebSocket to reconnect. You should then see a row for the WebSocket connection. If you click on it, you'll see a **Messages** tab where you can see all the messages passing through the WebSocket.

In the following example, you'll see the JavaScript code used to open the WebSocket connection and to send and receive messages:

script.js

```
const socket = new WebSocket('ws://localhost:8000/ws');  
  
// Connection opened  
socket.addEventListener('open', function (event) {  
  
    // Send message on form submission  
    document.getElementById('form').addEventListener('submit',
```

```
(event) => {
    event.preventDefault();
    const message = document.getElementById('message').value;

    addMessage(message, 'client');

    socket.send(message);

    event.target.reset();
});

// Listen for messages
socket.addEventListener('message', function (event) {
    addMessage(event.data, 'server');
});
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/echo/script.js>

As you can see, modern browsers provide a very simple API to interact with WebSockets. You just have to instantiate a new `WebSocket` object with the **Uniform Resource Locator (URL)** of your endpoint and wire some event listeners: `open` when the connection is ready and `message` when data is received from the server. Finally, the `send` method allows you to send data to the server. You can view more details on the `WebSocket` API in the **Mozilla Developer Network (MDN)** documentation at https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

Handling concurrency

In the previous example, we've assumed that the client was always sending a message first: we wait for its message before sending it back. Once again, it's the client that takes the initiative in the conversation.

However, in usual scenarios, the server can have data to send to the client without being at the initiative. In a chat application, another user can typically send one or several messages that we want to forward to the first user immediately. In this context, the blocking call to `receive_text` we showed in the previous example is a problem: while we are waiting, the server could have messages to forward to the client.

To solve this, we'll rely on more advanced tools of the `asyncio` module. Indeed, it provides functions that allow us to schedule several coroutines concurrently and wait until one of them is complete. In our context, we can have a coroutine that waits for client messages and another one that sends data to it when it arrives. The first one being fulfilled wins and we can start again with another loop iteration.

To make this clearer, let's build another example, in which the server will once again echo back the message of the client. Besides that, it'll regularly send the current time to the client. You can see the implementation in the following code snippet:

app.py

```
async def echo_message(websocket: WebSocket):
    data = await websocket.receive_text()
    await websocket.send_text(f"Message text was: {data}")

async def send_time(websocket: WebSocket):
    await asyncio.sleep(10)
    await websocket.send_text(f"It is: {datetime.utcnow().isoformat()}")

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    try:
        while True:
            echo_message_task = asyncio.create_task(echo_message(websocket))
            send_time_task = asyncio.create_task(send_time(websocket))
            done, pending = await asyncio.wait(
                {echo_message_task, send_time_task},
                return_when=asyncio.FIRST_COMPLETED,
            )
            for task in pending:
                task.cancel()
            for task in done:
```

```
task.result()
except WebSocketDisconnect:
    await websocket.close()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/concurrency/app.py>

As you can see, we defined two coroutines: the first one, `echo_message`, waits for text messages from the client and sends them back, while the second one, `send_time`, waits for 10 seconds before sending the current time to the client. Both of them expect a `WebSocket` instance in the argument.

The most interesting part lives under the infinite loop: as you can see, we call our two functions, wrapped by the `create_task` function of `asyncio`. This transforms the coroutine into a `Task` object. Under the hood, a task is how the event loop manages the execution of the coroutine. Put more simply, it gives us full control over the execution of the coroutine, to retrieve its result or even cancel it.

Those `task` objects are necessary to work with `asyncio.wait`. This function is especially useful to run tasks concurrently. It expects in the first argument a set of tasks to run. By default, this function will block until all given tasks are completed. However, we can control that thanks to the `return_when` argument: in our case, we want it to block until one of the tasks is completed, which corresponds to the `FIRST_COMPLETED` value. The effect is the following: our server will launch the coroutines concurrently. The first one will block waiting for a client message, while the other one will block for 10 seconds. If the client sends a message before 10 seconds, it'll send the message back and complete. Otherwise, the `send_time` coroutine will send the current time and complete.

At that point, `asyncio.wait` will return us two sets: the first one, `done`, contains a set of completed tasks, while the other one, `pending`, contains a set of tasks not yet completed.

We want to now go back to the start of the loop to start again. However, we need to first cancel all the tasks that have not been completed; otherwise, they would pile up at each iteration, hence the iteration over the `pending` set to `cancel` those tasks.

Finally, we also make an iteration over the `done` tasks and call the `result` method on them. This method returns the result of the coroutine but also re-raises an exception that could have been raised inside. This is especially useful to handle once again the disconnection of the client: when waiting for client data, if the tunnel is closed, an exception is raised. Thus, our `try...except` statement can catch it to properly close the `WebSocket`.

If you try this example as we did previously, you'll see that the server will regularly send you the current time but is also able to echo the messages you send.

This `send_time` example shows you how you can implement a process to send data to the client when an event happens on the server: new data is available in the database, an external process has finished a long computation, and so on. In the next section, we'll see how we can properly handle the case of multiple clients sending messages to the server, which then broadcasts them to all the clients.

That's basically how you can handle concurrency with `asyncio` tools. So far, everyone is able to connect to those WebSocket endpoints without any restriction. Of course, as with classic HTTP endpoints, you'll likely need to authenticate a user before opening the connection.

Using dependencies

Just as with regular endpoints, you can use dependencies in WebSocket endpoints. However, since they are designed with HTTP in mind, this comes with a few drawbacks.

First of all, you can't use security dependencies, as we showed in *Chapter 7, Managing Authentication and Security in FastAPI*. Indeed, under the hood, most of them work by injecting the `Request` object, which only works for HTTP requests (we saw that WebSockets are injected in a `WebSocket` object instead). Trying to inject those dependencies in a WebSocket context will result in an error.

Similarly, basic dependencies such as `Query`, `Header`, or `Cookie` have their quirks. Indeed, FastAPI is perfectly able to solve them in a WebSocket context. However, if they are required, FastAPI will throw an error when they are missing. Contrary to the HTTP validation error that is handled globally to render a proper 422 error, there is no handler for this WebSocket equivalent at the time of writing. This comes from a limitation of Starlette, the underlying server layer, that may be solved in future releases. You can follow the work on this subject at the following GitHub pull request: <https://github.com/encode/starlette/pull/527>.

Meanwhile, *it's recommended to make all your WebSocket dependencies optional* and handle missing values yourself.

That's what we'll see in our next example. In this one, we'll inject two dependencies, as follows:

- A `username` query parameter, which we'll use to greet the user on connection.

- A token cookie, which we'll compare with a static value, to keep the example simple. Of course, a proper strategy would be to have a proper user lookup, as we implemented in *Chapter 7, Managing Authentication and Security in FastAPI*. If this cookie is missing or doesn't have the required value, we'll close the WebSocket immediately with an error code.

Let's see the implementation in the following sample:

dependencies.py

```
@app.websocket("/ws")
async def websocket_endpoint(
    websocket: WebSocket,
    username: str = "Anonymous",
    token: Optional[str] = Cookie(None),
):
    if token != API_TOKEN:
        await websocket.close(code=status.WS_1008_POLICY_VIOLATION)
        return

    await websocket.accept()
    await websocket.send_text(f"Hello, {username}!")
    try:
        while True:
            data = await websocket.receive_text()
            await websocket.send_text(f"Message text was: {data}")
    except WebSocketDisconnect:
        await websocket.close()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/dependencies/app.py>

As you can see, injecting dependencies is no different from standard HTTP endpoints. Notice that we take care of providing a default value or making them optional, as we said before.

Then, we can have our dummy authentication logic. If it fails, we immediately close the socket with a status code. WebSockets have their own set of status codes. You can view a complete list of these on this MDN documentation page: <https://developer.mozilla.org/fr/docs/Web/API/CloseEvent>. The most generic one when an error occurs is 1008.

If it passes, we can start our classic echo server. Notice that we can use the username value as we wish in our logic. Here, we send a first message to greet the user on connection. If you try this with the HTML application, you'll see this message first, as shown in the following screenshot:

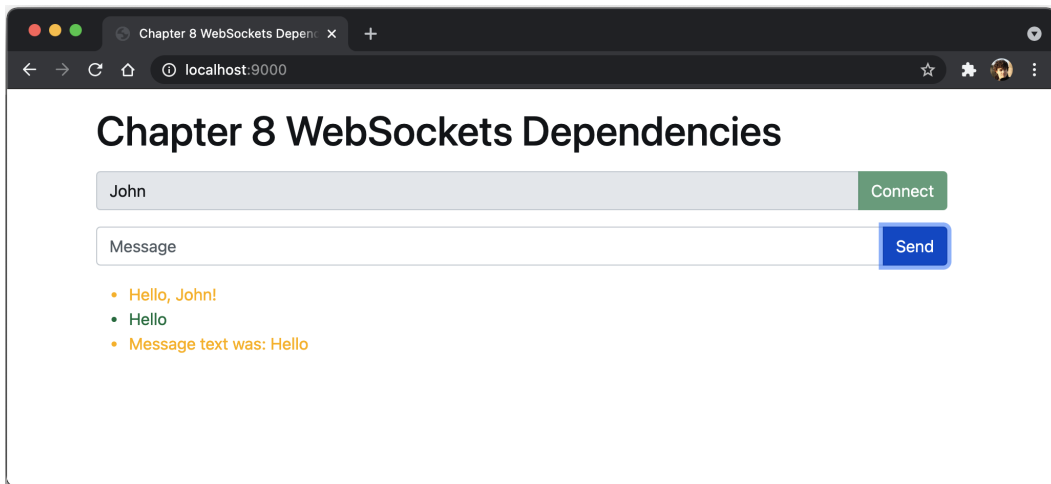


Figure 8.2 – Greeting message on connection

With the browser WebSocket API, query parameters can be passed into the URL and the browser automatically forwards the cookies. However, there is *no way to pass custom headers*. This means that if you rely on headers for authentication, you'll have to either add one using cookies or implement an authentication message mechanism in the WebSocket logic itself. However, if you don't plan to use your WebSocket with a browser, you can still rely on headers since most WebSocket clients support them.

You now have a good overview of how to add WebSockets to your FastAPI application. As we said, they are generally useful when several users are involved in real time and we need to broadcast messages to all of them. We'll see in the next section how to implement this pattern reliably.

Handling multiple WebSocket connections and broadcasting messages

As we said in the introduction to this chapter, a typical use case for WebSockets is to implement real-time communication across multiple clients, such as a chat application. In this configuration, several clients have an open WebSocket tunnel with the server. Thus, the role of the server is to *manage all the client connections and broadcast messages to all of them*: when a user sends a message, the server has to send it to all other clients in their WebSockets. We show you a schema of this principle here:

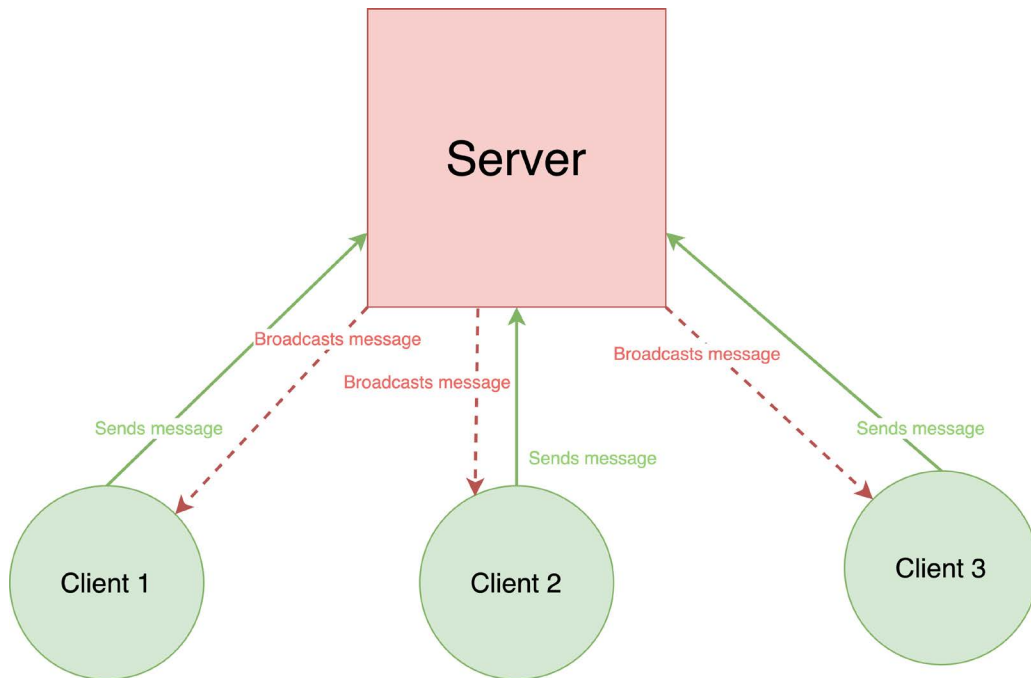


Figure 8.3 – Multiple clients connected through WebSocket to a server

A first approach could be simply to keep a list of all WebSocket connections and iterate through them to broadcast messages. This would work but would quickly become problematic in a production environment. Indeed, most of the time, server processes run multiple workers when deployed. This means that instead of having only one process serving requests, we can have several ones so that we can answer more requests concurrently. We could also think of deployments on multiple servers spread in several data centers.

Hence, nothing guarantees you that two clients opening a WebSocket are served by the same process. Our simple approach would fail in this configuration: since connections are kept in the process memory, the process receiving the message would not be able to broadcast the message to clients served by another process. We schematize this problem in the following diagram:

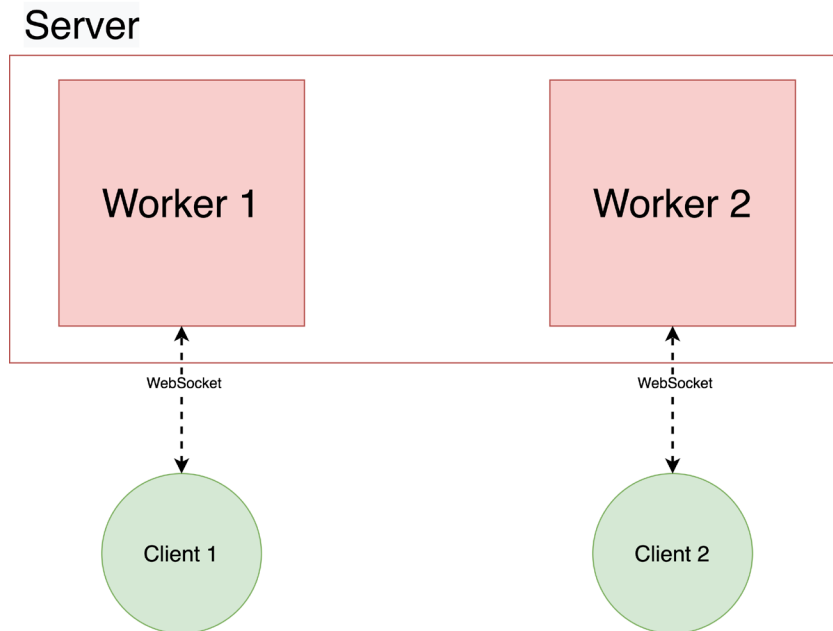


Figure 8.4 – Multiple server workers without a message broker

To solve this, we generally rely on **message brokers**. Message brokers are pieces of software whose role is to receive messages published by a first program and broadcast them to programs that are subscribed to it. Usually, this **publish-subscribe (pub-sub)** pattern is organized into different channels so that messages are clearly organized following their topic or usage. Some of the best-known message broker software includes Apache Kafka, RabbitMQ, or cloud-based implementations from **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)** and **Microsoft Azure: Amazon MQ, Cloud Pub/Sub and Service Bus**, respectively.

Hence, our message broker will be unique in our architecture, and several server processes will connect to it to either publish or subscribe to messages. This architecture is schematized in the following diagram:

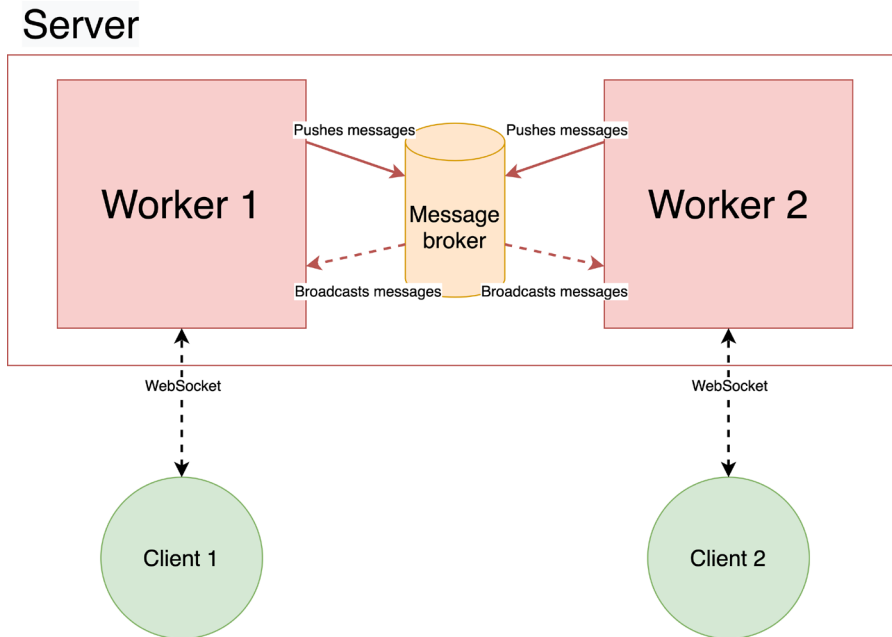


Figure 8.5 – Multiple server workers with a message broker

In this chapter, we'll see how to set up a simple system using the `broadcaster` library from Encode (the creators of Starlette) and `Redis`, which will act as a message broker.

A word on Redis

As its core, Redis is a data store designed to achieve maximum performance. It's widely used in the industry for storing temporary data that we want to access very quickly, such as cache or distributed locks. It also supports a basic pub/sub paradigm, which makes it a good candidate to be used as a message broker. You can learn more about this technology at its official website <https://redis.io>.

First of all, let's install the library with the following command:

```
$ pip install "broadcaster[redis]"
```

This library will abstract away all the complexities of publishing and subscribing with Redis for us.

Let's see the details of the implementation. In the following example, you'll see the instantiation of the `Broadcaster` object:

app.py

```
broadcast = Broadcast("redis://localhost:6379")
CHANNEL = "CHAT"
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/broadcast/app.py>

As you can see, it only expects a URL to our Redis server. Notice also that we define a `CHANNEL` constant. This will be the name of the channel to publish and subscribe to messages. We choose a static value here for the sake of the example, but you could have dynamic channel names in a real-world application—to support several chat rooms, for example.

Then, we define two functions: one to subscribe to new messages and send them to the client and another one to publish messages received in the `WebSocket`. You can see these functions in the following sample:

app.py

```
class MessageEvent(BaseModel):
    username: str
    message: str

async def receive_message(websocket: WebSocket, username: str):
    async with broadcast.subscribe(channel=CHANNEL) as subscriber:
        async for event in subscriber:
            message_event = MessageEvent.parse_raw(event.message)
            # Discard user's own messages
            if message_event.username != username:
                await websocket.send_json(message_event.dict())

async def send_message(websocket: WebSocket, username: str):
```

```
data = await websocket.receive_text()
event = MessageEvent(username=username, message=data)
await broadcast.publish(channel=CHANNEL, message=event.json())
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/broadcast/app.py>

First of all, notice that we defined a Pydantic model, `MessageEvent`, to help us structure the data contained in a message. Instead of just passing raw strings as we've been doing up to now, we have an object bearing both the message and the username.

The first function, `receive_message`, subscribes to the broadcast channel and waits for messages called `event`. The data of the message contains serialized **JavaScript Object Notation (JSON)** that we deserialize to instantiate a `MessageEvent` object. Notice that we use the `parse_raw` method of the Pydantic model, allowing us to parse the JSON string into an object in one operation.

Then, we check if the message username is different from the current username. Indeed, since all users are subscribed to the channel, they will also receive the messages they sent themselves. That's why we discard them based on the username to avoid this. Of course, in a real-world application, you'll likely want to rely on a unique **user identifier (UID)** rather than a simple username.

Finally, we can send the message through the WebSocket thanks to the `send_json` method, which takes care of serializing the dictionary automatically.

The second function, `send_message`, is there to publish a message to the broker. Quite simply, it waits for new data in the socket, structures it into a `MessageEvent` object, and then publishes it.

That's about it for the *broadcaster* part. We then have the WebSocket implementation in itself, which is very similar to what we saw in the previous sections. You can see it in the following sample:

app.py

```
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket, username:
str = "Anonymous"):
    await websocket.accept()
    try:
        while True:
```

```
        receive_message_task = asyncio.create_task(
            receive_message(websocket, username)
        )
        send_message_task = asyncio.create_task(
            send_message(websocket, username)
        )
        done, pending = await asyncio.wait(
            {receive_message_task, send_message_task},
            return_when=asyncio.FIRST_COMPLETED,
        )
        for task in pending:
            task.cancel()
        for task in done:
            task.result()
    except WebSocketDisconnect:
        await websocket.close()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/broadcast/app.py>

Notice that username is retrieved from the query parameters.

Finally, we need to tell FastAPI to open the connection with the broker when it starts the application and to close it when exiting, as you can see in the following extract:

app.py

```
@app.on_event("startup")
async def startup():
    await broadcast.connect()

@app.on_event("shutdown")
async def shutdown():
    await broadcast.disconnect()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter8/broadcast/app.py>

The `on_event` decorators allow us to trigger some useful logic when FastAPI starts or stops.

Let's now try this application! First, we'll run the Uvicorn server. Be sure that your Redis container is running before starting, as we explained in the *Technical requirements* section. Here's the code you'll need:

```
$ uvicorn chapter8.broadcast.app:app
```

We also provided a simple HTML client in the examples. To run it, we can simply serve it with the built-in Python server, as follows:

```
$ python -m http.server --directory chapter8/broadcast 9000
```

You can now access it through `http://localhost:9000`. If you open it twice in your browser, in two different windows, you can see whether the broadcasting is working. Input a username in the first window and click on **Connect**. Do the same in the second window with a different username. You can now send messages and see that they are broadcasted to the other client, as depicted in the following screenshot:

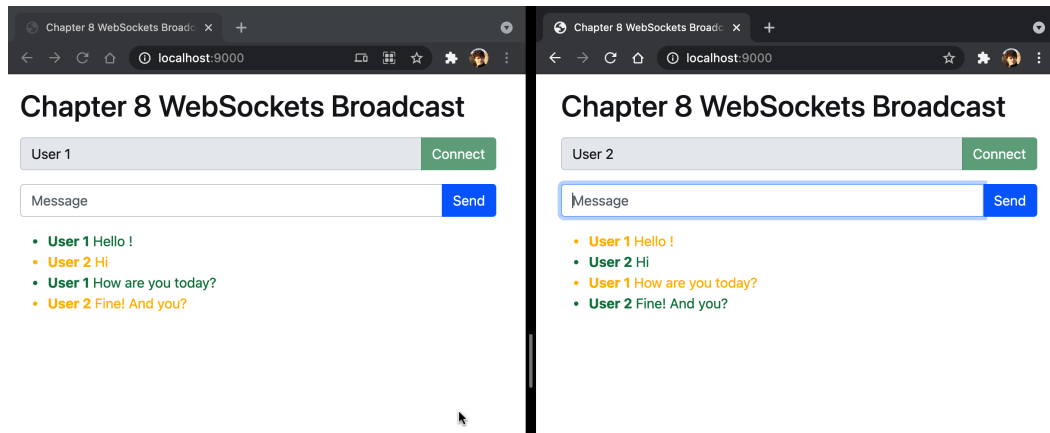


Figure 8.6 – Multiple WebSocket clients broadcasting messages

That was a very quick overview of how you can implement broadcasting systems involving message brokers. Of course, we only covered the basics here, and much more complex things can be done with those powerful technologies. Once again, we see that FastAPI gives us access to powerful building bricks without locking us inside specific technologies or patterns: it's very easy to include new libraries to expand our possibilities.

Summary

In this chapter, you've learned how to work with one of the latest web technologies available: WebSocket. You are now able to open a two-way communication channel between a client and a server, allowing you to implement applications with real-time constraints. As you've seen, FastAPI makes it very easy to add such endpoints. Still, the way of thinking inside a WebSocket logic is quite different from traditional HTTP endpoints: managing an infinite loop and handling several tasks at a time are completely new challenges. Fortunately, the asynchronous nature of the framework makes our life easier in this matter and helps us write concurrent code that is easily understandable.

Finally, we also had a quick overview of the challenges to solve when handling multiple clients that share messages between them. You saw that message broker software such as Apache Kafka or RabbitMQ is necessary to make this use case reliable across several server processes.

You are now acquainted with all the features of FastAPI. Up to now, we've shown very simple examples focused on a specific point. In the real world, however, you'll likely develop big applications that can do a lot of things and grow larger over time. To make them reliable, maintainable, and keep high-quality code, it's necessary to test them to make sure they behave as intended and that you don't introduce bugs when adding new things.

In the next chapter, you'll see how to set up an efficient test environment for FastAPI.

9

Testing an API Asynchronously with pytest and HTTPX

In software development, a significant part of the developer's work should be dedicated to writing tests. At first, you may be tempted to manually test your application by running it, making a few requests, and arbitrarily deciding that "everything works". However, this approach is flawed and can't guarantee that your program works in every circumstance and that you didn't break things along the way.

That's why several disciplines have emerged regarding software testing: unit tests, integration tests, E2E tests, acceptance tests, and more. These techniques aim to validate the functionality of the software from a micro level, where we test single functions (unit tests), to a macro level, where we test a global feature that delivers value to the user (acceptance tests). In this chapter, we'll focus on the first level: unit testing.

Unit tests are short programs designed to verify that our code behaves the way it should in every circumstance. You may think that tests are time-consuming to write and that they don't add value to your software, but this will save you time in the long run: first of all, tests can be run automatically in a few seconds, ensuring that all your software works, without you needing to manually go over every feature. Secondly, when you introduce new features or refactor the code, you're ensuring that you don't introduce bugs to existing parts of the software. In conclusion, tests are just as important as the program itself, and they help you deliver reliable and high-quality software.

In this chapter, you'll learn how to write tests for your FastAPI application, both for HTTP endpoints and WebSockets. To help with this, you'll learn how to configure pytest, a well-known Python test framework, and HTTPX, an asynchronous HTTP client for Python.

In this chapter, we're going to cover the following main topics:

- Introduction to unit testing with pytest
- Setting up the testing tools for FastAPI with HTTPX
- Writing tests for REST API endpoints
- Writing tests for WebSocket endpoints

Technical requirements

For this chapter, you'll need a Python virtual environment, similar to the one we set up in *Chapter 1, Python Development Environment Setup*.

For the *Testing with a database* section, you'll need a running MongoDB server on your local computer. The easiest way to do this is to run it as a Docker container. If you've never used Docker before, we recommend that you read the Get Started tutorial in the official documentation: <https://docs.docker.com/get-started/>. Once done, you'll be able to run a MongoDB server with this simple command:

```
$ docker run -d --name fastapi-mongo -p 27017:27017 mongo:4.4
```

The MongoDB server instance will then be available on your local computer on port 27017.

You can find all the code examples for this chapter in its dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter9>.

Introduction to unit testing with pytest

As we mentioned in the introduction, writing unit tests is an essential task in software development to deliver high-quality software. To help us be productive and efficient, lot of libraries exist that provide tools and shortcuts dedicated to testing. In the Python standard library, a module exists for unit testing called `unittest`. Even though it's quite common in Python code bases, many Python developers tend to prefer `pytest`, which provides a more lightweight syntax and powerful tools for advanced use cases.

In the following examples, we'll write a unit test for a function called `add`, both with `unittest` and `pytest`, so that you can see how they compare on a basic use case. First, we'll install `pytest`:

```
$ pip install pytest
```

Now, let's see our simple `add` function, which simply performs an addition:

chapter9_introduction.py

```
def add(a: int, b: int) -> int:
    return a + b
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_introduction.py

Now, let's implement a test that checks that `2 + 3` is indeed equal to `5` with `unittest`:

chapter9_introduction_unittest.py

```
import unittest

from chapter9.chapter9_introduction import add

class TestChapter9Introduction(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_introduction_unittest.py

As you can see, `unittest` expects us to define a class inheriting from `TestCase`. Then, each test lives in its own method. To assert that two values are equal, we must use the `assertEqual` method.

To run this test, we can call the `unittest` module from the command line and pass it through the dotted path to our test module:

```
$ python -m unittest chapter9.chapter9_introduction_unittest
.
-----
-----
Ran 1 test in 0.000s

OK
```

In the output, each successful test is represented by a dot. If one or several tests are not successful, you will get a detailed error report for each, highlighting the failing assertion. You can try it by changing the assertion in the test.

Now, let's write the same test with `pytest`:

chapter9_introduction_unittest.py

```
from chapter9.chapter9_introduction import add

def test_add():
    assert add(2, 3) == 5
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_introduction_unittest.py

As you can see, it's much shorter! Indeed, with `pytest`, you don't necessarily have to define a class: a simple function is enough. The only constraint to making it work is that the function name has to start with `test_`. This way, `pytest` can automatically discover the test functions. Secondly, it relies on the built-in `assert` statement instead of specific methods, allowing you to write comparisons more naturally.

To run this test, we must simply call the `pytest` executable with the path to our test file:

```
$ pytest chapter9/chapter9_introduction_pytest.py
===== test session starts =====
platform darwin -- Python 3.7.10, pytest-6.2.4, py-1.10.0,
pluggy-0.13.1
rootdir: /Users/fvoron/Google Drive/Livre FastAPI/Building-
Data-Science-Applications-with-FastAPI, configfile: setup.cfg
plugins: asyncio-0.15.1, cov-2.12.0, mock-3.6.1, repeat-0.9.1
collected 1 item

chapter9/chapter9_introduction_pytest.py .          [100%]

===== 1 passed in 0.01s =====
```

Once again, the output represents each successful test with a dot. Of course, if you change the test to make it fail, you'll get a detailed error for the failing assertion.

It's worth noting that if you run `pytest` without any arguments, it'll automatically discover all the test files living in your folder, as long as their name starts with `test_`.

Here, we made a small comparison between `unittest` and `pytest`. For the rest of this chapter, we'll stick with `pytest`, which should give you a more productive experience while writing tests.

At the beginning of this section, we said that `pytest` provides powerful tools to help us write tests. Before focusing on FastAPI testing, we'll review two of them: `parametrize` and `fixtures`.

Generating tests with `parametrize`

In our previous example with the `add` function, we only tested one addition test, `2 + 3`. Most of the time, we'll want to check for more cases to ensure our function works in every circumstance. Our first approach could be to add more assertions to our test, like so:

```
def test_add():
    assert add(2, 3) == 5
    assert add(0, 0) == 0
    assert add(100, 0) == 100
    assert add(1, 1) == 2
```

While working, this method has two drawbacks: first, it may be a bit cumbersome to write the same assertion several times with only some parameters changing. In this example, it's not too bad, but tests can be way more complex, as we'll see with FastAPI. Secondly, we still only have one test: the first failing assertion will stop the test and the following ones won't be executed. Thus, we'll only know the result if we fix the failing assertion first and run the test again.

To help with this specific task, `pytest` provides the `parametrize` marker. In `pytest`, a **marker** is a special decorator that's used to easily pass metadata to the test. Special behaviors can then be implemented, depending on the markers used by the test.

Here, `parametrize` allows us to pass several sets of variables that will be passed as arguments to the test function. At runtime, each set will generate a new and independent test. To understand this better, let's look at how to use this marker to generate several tests for our `add` function:

chapter9_introduction_pytest_parametrize.py

```
import pytest

from chapter9.chapter9_introduction import add

@pytest.mark.parametrize("a,b,result", [(2, 3, 5), (0, 0, 0),
(100, 0, 100), (1, 1, 2)])

def test_add(a, b, result):
    assert add(a, b) == result
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_introduction_pytest_parametrize.py

Here, you can see that we simply decorated our test function with the `parametrize` marker. The basic usage is as follows: the first argument is a string with the name of each parameter separated by a comma. Then, the second argument is a list of tuples. Each tuple contains the values of the parameters in order.

Our test function receives those parameters in arguments, each one named the way you specified previously. Thus, you can use them at will in the test logic. As you can see, the great benefit here is that we only have to write the `assert` statement once. Besides, it's very quick to add a new test case: we just have to add another tuple to the `parametrize` marker.

Now, let's run this test to see what happens by using the following command:

```
$ pytest chapter9/chapter9_introduction_pytest_parametrize.py
===== test session starts =====
platform darwin -- Python 3.7.10, pytest-6.2.4, py-1.10.0,
pluggy-0.13.1
rootdir: /Users/fvoron/Google Drive/Livre FastAPI/Building-
Data-Science-Applications-with-FastAPI, configfile: setup.cfg
plugins: asyncio-0.15.1, cov-2.12.0, mock-3.6.1, repeat-0.9.1
collected 4 items

chapter9/chapter9_introduction_pytest_parametrize.py ....
[100%]

===== 4 passed in 0.01s =====
```

As you can see, `pytest` executed *four tests instead of one!* This means that it generated four independent tests, along with their own sets of parameters. If several tests are failing, we'll be informed, and the output will tell us which set of parameters caused the error.

To conclude, `parametrize` is a very convenient way to test different outcomes when it's given a different set of parameters.

While writing unit tests, you'll often need variables and objects several times across your tests, such as in an app instance, as some fake data, and so on. To avoid having to repeat the same things over and over across your tests, `pytest` proposes an interesting feature: fixtures.

Reusing test logic by creating fixtures

When testing a large application, tests tend to become quite repetitive: lots of them will share the same boilerplate code before their actual assertion. Let's consider using Pydantic models to represent a person and their postal address:

`chapter9_introduction_fixtures.py`

```
from datetime import date
from enum import Enum
from typing import List
```

```
from pydantic import BaseModel

class Gender(str, Enum):
    MALE = "MALE"
    FEMALE = "FEMALE"
    NON_BINARY = "NON_BINARY"

class Address(BaseModel):
    street_address: str
    postal_code: str
    city: str
    country: str

class Person(BaseModel):
    first_name: str
    last_name: str
    gender: Gender
    birthdate: date
    interests: List[str]
    address: Address
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_introduction_fixtures.py

This example may look familiar: it was taken from *Chapter 4, Managing pydantic Data Models in FastAPI*. Now, let's say that we want to write tests with some instances of those models. Obviously, it would be a bit annoying to instantiate them in each test, filling them with fake data.

Fortunately, fixtures allow us to write them in one go. The following example shows how to use them:

chapter9_introduction_fixtures_test.py

```
import pytest

from chapter9.chapter9_introduction_fixtures import Address,
Gender, Person
```

```
@pytest.fixture
def address():
    return Address(
        street_address="12 Squirell Street",
        postal_code="424242",
        city="Woodtown",
        country="US",
    )

@pytest.fixture
def person(address):
    return Person(
        first_name="John",
        last_name="Doe",
        gender=Gender.MALE,
        birthdate="1991-01-01",
        interests=["travel", "sports"],
        address=address,
    )

def test_address_country(address):
    assert address.country == "US"

def test_person_first_name(person):
    assert person.first_name == "John"

def test_person_address_city(person):
    assert person.address.city == "Woodtown"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_introduction_fixtures_test.py

Once again, pytest makes it very straightforward: fixtures are *simple functions* decorated with the *fixture decorator*. Inside, you can write any logic and return the data you'll need in your tests. Here, in `address`, we instantiate an `Address` object with fake data and return it.

Now, how can we use this fixture? If you look at the `test_address_country` test, you'll see some magic happening: by setting an `address` argument on the test function, pytest automatically detects that it corresponds to the `address` fixture, executes it, and passes its return value. Inside the test, we have our `Address` object ready to use. pytest calls this *requesting a fixture*.

You may have noticed that we also defined another fixture, `person`. Once again, we instantiate a `Person` model with dummy data. The interesting thing to note, however, is that we actually requested the `address` fixture to use it inside! That's what makes this system so powerful: fixtures can depend on other fixtures, which can also depend on others, and so on. In some way, it's quite similar to dependency injection, as we discussed in *Chapter 5, Dependency Injections in FastAPI*.

With that, our quick introduction to pytest has come to an end. Of course, there are so many more things to say, but this will be enough for you to get started. If you want to explore this topic further, you can read the official pytest documentation, which includes tons of examples showing you how you can benefit from all its features: <https://docs.pytest.org/en/latest/>.

Now, let's focus on FastAPI. We'll start by setting up the tools for testing our applications.

Setting up testing tools for FastAPI with HTTPX

If you look at the FastAPI documentation regarding testing, you'll see that it recommends that you use `TestClient` provided by Starlette. In this book, we'll show you a different approach involving an HTTP client, called HTTPX.

Why? The default `TestClient` is implemented in a way that makes it completely synchronous, meaning you can write tests without worrying about `async` and `await`. This might sound nice, but we found that it causes some problems in practice: since your FastAPI app is designed to work asynchronously, you'll likely have lots of services working asynchronously, such as the database drivers we saw in *Chapter 6, Databases and Asynchronous ORMs*. Thus, in your tests, you'll probably need to perform some actions on those asynchronous services, such as filling a database with dummy data, which will make your tests asynchronous anyway. Melting the two approaches often leads to strange errors that are hard to debug.

Fortunately, HTTPX, an HTTP client created by the same team as Starlette, allows us to have a pure asynchronous HTTP client able to make requests to our FastAPI app. To make this approach work, we'll need three libraries:

- HTTPX, the client that will perform HTTP requests

- `asgi-lifespan`, a library for managing the startup and shutdown events of your FastAPI app programmatically
- `pytest-asyncio`, an extension of `pytest` that allows us to write asynchronous tests

Let's install these libraries using the following command:

```
$ pip install httpx asgi-lifespan pytest-asyncio
```

Great! Now, let's write some fixtures so that we can easily get an HTTP test client for a FastAPI application. This way, when writing a test, we'll only have to request the fixture and we'll be able to make a request right away.

In the following example, we are considering a simple FastAPI application that we want to test:

chapter9_app.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def hello_world():
    return {"hello": "world"}

@app.on_event("startup")
async def startup():
    print("Startup")

@app.on_event("shutdown")
async def shutdown():
    print("Shutdown")
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app.py

In a separate test file, we'll implement two fixtures.

The first one, `event_loop`, will ensure that we always work with the same event loop instance. It's automatically requested by `pytest-asyncio` before executing asynchronous tests. While not strictly required, experience has shown us that it greatly helps us avoid errors that may occur when several event loops are launched. You can see its implementation in the following example:

chapter9_app_test.py

```
@pytest.fixture(scope="session")
def event_loop():
    loop = asyncio.get_event_loop()
    yield loop
    loop.close()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_test.py

Here, you can see that we simply get the current event loop before *yielding* it. As we discussed in *Chapter 2, Python Programming Specificities*, using a generator allows us to "pause" the function's execution and get back to the execution of its caller. This way, when the caller is done, we can execute cleanup operations, such as closing the loop. `pytest` is smart enough to handle this correctly in fixtures, so this is a very common pattern for setting up test data, using it, and destroying it after.

Of course, this function is decorated with the `fixture` decorator to make it a fixture for `pytest`. You may have noticed that we added an argument called `scope` with a value of `session`. This argument controls at which level the fixture should be instantiated. By default, it's recreated *at the beginning of each single test function*. The `session` value is the highest level, meaning that the fixture is only created once at the beginning of the whole test run, which is relevant for our event loop. You can find out more about this more advanced feature in the official documentation: <https://docs.pytest.org/en/latest/how-to/fixtures.html#scope-sharing-fixtures-across-classes-modules-packages-or-session>.

Next, we'll implement our `test_client` fixture, which will create an instance of HTTPX for our FastAPI application. We must also remember to trigger the app events with `asgi-lifespan`. You can see what it looks like in the following example:

chapter9_app_test.py

```
@pytest.fixture
async def test_client():
    async with LifespanManager(app):
        async with httpx.AsyncClient(app=app, base_url="http://
app.io") as test_client:
            yield test_client
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_test.py

Only three lines are needed. Notice that the `app` variable is our FastAPI application instance is the one we imported from its module, from `chapter9.chapter9_app`

```
import app.
```

Up until now, we haven't had the opportunity to talk about the `with` syntax. In Python, this is what's called a **context manager**. Simply put, it's a convenient syntax for objects that need to execute *setup logic* when they are used and *teardown logic* when they are not needed anymore. When you *enter* the `with` block, the object automatically executes the setup logic. When you *exit* the block, it executes its teardown logic. You can read more about context managers in the Python documentation: <https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers>.

In our case, both `LifespanManager` and `httpx.AsyncClient` work as context managers, so we simply have to nest their blocks. The first one ensures startup and shutdown events are executed, while the second one ensures that an HTTP session is ready.

Notice that we once again used a generator here, with `yield`. This is important because, even if we don't have any more code after, we have to *give the context managers the opportunity to exit*: after the `yield` statement, we implicitly exit the `with` blocks.

That's it! We now have all the fixtures ready to write tests for our REST API endpoints. That's what we'll do in the next section.

Organizing tests and global fixtures in projects

In larger projects, you'll likely have several test files to keep your tests organized. Usually, those files are placed in a `tests` folder at the root of your project. If your test files are prefixed with `test_`, they will be automatically discovered by pytest. *Figure 9.1* shows an example of this.

Besides this, you'll need the fixtures we defined in this section for all your tests. Rather than repeating them again and again in all your test files, pytest allows you to write global fixtures in a file named `conftest.py`. After putting it in your `tests` folder, it will automatically be imported, allowing you to request all the fixtures you define inside it. You can read more about this in the official documentation at <https://docs.pytest.org/en/latest/reference/fixtures.html#conftest-py-sharing-fixtures-across-multiple-files>:

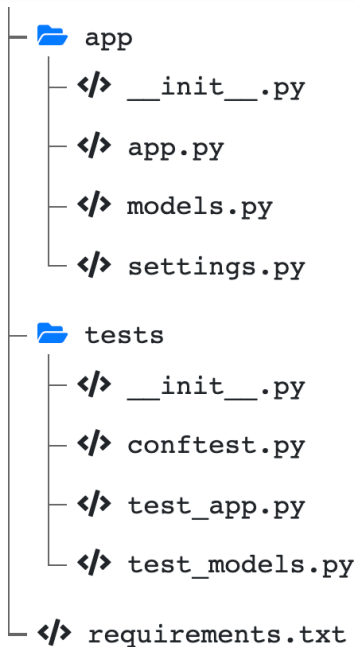


Figure 9.1 – Structure of a project containing tests

Writing tests for REST API endpoints

All the tools we need to test our FastAPI application are now ready. All these tests boil down to performing an HTTP request and checking the response to see if it corresponds to what we expect.

Let's start simple with a test for our `hello_world` path operation function. You can see it in the following code:

chapter9_app_test.py

```
@pytest.mark.asyncio
async def test_hello_world(test_client: httpx.AsyncClient):
    response = await test_client.get("/")

    assert response.status_code == status.HTTP_200_OK

    json = response.json()
    assert json == {"hello": "world"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_test.py

First of all, notice that the test function is defined as `async`. As we mentioned previously, to make it work with `pytest`, we had to install `pytest-asyncio`. This extension provides the `asyncio` marker: each asynchronous test should be decorated with this marker to make it work properly.

Next, we requested our `test_client` fixture, which we defined earlier. It gives us an `HTTPX` client instance ready to make requests to our FastAPI app. Note that we manually type hinted the fixture. While not strictly required, it'll greatly help you if you use an IDE such as Visual Studio Code, which uses type hints to provide you with convenient auto-completion features.

Then, in the body of our test, we performed the request. Here, it's a simple GET request to the `/` path. It returns an `HTTPX Response` object (which is *different* from the `Response` class of FastAPI) containing all the data of the HTTP response: the status code, the headers, and the body.

Finally, we made assertions based on this data. As you can see, we verified that the status code was indeed 200. We also checked the content of the body, which is a simple JSON object. Notice that the `Response` object has a convenient method called `json` for automatically parsing JSON content.

Great! We wrote our first FastAPI test! Of course, you'll likely have more complex tests, typically ones for POST endpoints.

Writing tests for POST endpoints

Testing a POST endpoint is not very different from what we've seen earlier. The difference is that we'll likely have more cases to check if data validation is working. In the following example, we are implementing a POST endpoint that accepts a `Person` model in the body:

`chapter9_app_post.py`

```
class Person(BaseModel):
    first_name: str
    last_name: str
    age: int

@app.post("/persons", status_code=status.HTTP_201_CREATED)
async def create_person(person: Person):
    return person
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_post.py

An interesting test could be to ensure that an error is raised if some fields are missing in the request payload. In the following extract, we wrote two tests: one with an invalid payload and another with a valid one:

`chapter9_app_post_test.py`

```
@pytest.mark.asyncio
class TestCreatePerson:
    async def test_invalid(self, test_client: httpx.
```

```
AsyncClient):
    payload = {"first_name": "John", "last_name": "Doe"}
    response = await test_client.post("/persons",
    json=payload)

    assert response.status_code == status.HTTP_422_
    UNPROCESSABLE_ENTITY

    async def test_valid(self, test_client: httpx.AsyncClient):
        payload = {"first_name": "John", "last_name": "Doe",
        "age": 30}
        response = await test_client.post("/persons",
        json=payload)

        assert response.status_code == status.HTTP_201_CREATED

        json = response.json()
        assert json == payload
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_post_test.py

The first thing you may have noticed is that we wrapped our two tests inside a class. While not required in pytest, it could help you organize your tests; for example, to regroup tests that concern a single endpoint. Notice that, in this case, we only have to decorate the class with the `asyncio` marker; it will be automatically applied on single tests. Also, ensure that you add the `self` argument to each test: since we are now inside a class, they become methods.

These tests are not very different from our first example. As you can see, the HTTPX client makes it very easy to perform POST requests with a JSON payload: you just have to pass a dictionary to the `json` argument.

Of course, HTTPX helps you build all kinds of HTTP requests with headers, query parameters, and so on. Be sure to check its official documentation to learn more about its usage: <https://www.python-httpx.org/quickstart/>.

Testing with a database

Your application will likely have a database connection to read and store data. In this context, you'll need to work with a fresh test database at each run to have a clean and predictable set of data to write your tests.

For this, we'll use two things. The first one, `dependency_overrides`, is a FastAPI feature that allows us to replace some dependencies at runtime. For example, we can replace the dependency that returns the database instance with another one that returns a test database instance. The second one is, once again, `fixtures`, which will help us create fake data in the test database before we run the tests.

To show you a working example, we'll consider the same example we built in the *Communicating with a MongoDB database with Motor* section of *Chapter 6, Databases and Asynchronous ORMs*. In this example, we built REST endpoints to manage blog posts. As you may recall, we had a `get_database` dependency that returned the database instance. As a reminder, we'll show it again here:

app.py

```
motor_client = AsyncIOMotorClient("mongodb://localhost:27017")
database = motor_client["chapter6_mongo"]

def get_database() -> AsyncIOMotorDatabase:
    return database
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter6/mongodb/app.py>

Path operation functions and other dependencies would then use this dependency to retrieve the database instance.

For our tests, we'll create a new instance of `AsyncIOMotorDatabase` that points to another database. Then, we'll create a new dependency, directly in our test file, that returns this instance. You can see this in the following example:

chapter9_db_test.py

```
motor_client = AsyncIOMotorClient("mongodb://localhost:27017")
database_test = motor_client["chapter9_db_test"]

def get_test_database():
    return database_test
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_db_test.py

Then, in our `test_client` fixture, we'll override the default `get_database` dependency by using our current `get_test_database` dependency. The following example shows how this is done:

chapter9_db_test.py

```
@pytest.fixture
async def test_client():
    app.dependency_overrides[get_database] = get_test_database
    async with LifespanManager(app):
        async with httpx.AsyncClient(app=app, base_url="http://
app.io") as test_client:
            yield test_client
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_db_test.py

FastAPI provides a property called `dependency_overrides`, which is a dictionary that maps original dependency functions with substitutes. Here, we directly used the `get_database` function as a key. The rest of the fixture doesn't have to change. Now, whenever the `get_database` dependency is injected into the application code, FastAPI will automatically replace it with `get_test_database`. As a result, our endpoints will now work with the test database instance.

To test some behaviors, such as retrieving a single post, it's usually convenient to have some base data in our test database. To allow this, we'll create a new fixture that will instantiate dummy `PostDB` objects and insert them into the test database. You can see this in the following example:

chapter9_db_test.py

```
@pytest.fixture(autouse=True, scope="module")
async def initial_posts():
    initial_posts = [
        PostDB(title="Post 1", content="Content 1"),
        PostDB(title="Post 2", content="Content 2"),
        PostDB(title="Post 3", content="Content 3"),
    ]
    await database_test["posts"].insert_many(
        [post.dict(by_alias=True) for post in initial_posts]
    )

    yield initial_posts

    await motor_client.drop_database("chapter9_db_test")
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_db_test.py

Here, you can see that we just had to make an `insert_many` request to the MongoDB database to create the posts.

Notice that we used the `autouse` and `scope` arguments of the fixture decorator. The first one tells pytest to automatically call this fixture *even if it's not requested in any test*. In this case, it's convenient because we'll always ensure that the data has been created in the database, without the risk of forgetting to request it in the tests. The other one, `scope`, allows us, as we mentioned previously, to not run this fixture at the beginning of each test. With the `module` value, the fixture will create the objects only once, at the beginning of this particular test file. It helps us keep the test fast because in this case, it doesn't make sense to recreate the posts before each test.

Once again, we *yield* the posts instead of returning them. This pattern allows us to delete the test database after the tests run. By doing this, we're making sure that we always start with a fresh database when we've run the tests.

And we are done! We can now write tests while knowing exactly what we have in the database. In the following example, you can see tests that are used to verify the behavior of the endpoint retrieving a single post:

chapter9_db_test.py

```
@pytest.mark.asyncio
class TestGetPost:
    async def test_not_existing(self, test_client: httpx.
AsyncClient):
        response = await test_client.get("/posts/abc")

        assert response.status_code == status.HTTP_404_NOT_
FOUND

    async def test_existing(
        self, test_client: httpx.AsyncClient, initial_posts:
List[PostDB]
    ):
        response = await test_client.get(f"/posts/{initial_
posts[0].id}")

        assert response.status_code == status.HTTP_200_OK

        json = response.json()
        assert json["_id"] == str(initial_posts[0].id)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_db_test.py

Notice that we requested the `initial_posts` fixture in the second test to retrieve the identifier of the truly existing post in our database.

Of course, we can also test our endpoints by creating data and checking if they correctly insert this data into the database. You can see this in the following example:

chapter9_db_test.py

```
@pytest.mark.asyncio
class TestCreatePost:
    async def test_invalid_payload(self, test_client: httpx.AsyncClient):
        payload = {"title": "New post"}
        response = await test_client.post("/posts",
            json=payload)

        assert response.status_code == status.HTTP_422_UNPROCESSABLE_ENTITY

    async def test_valid_payload(self, test_client: httpx.AsyncClient):
        payload = {"title": "New post", "content": "New post content"}
        response = await test_client.post("/posts",
            json=payload)

        assert response.status_code == status.HTTP_201_CREATED

        json = response.json()
        post_id = ObjectId(json["_id"])
        post_db = await database_test["posts"].find_one({"_id":
            post_id})
        assert post_db is not None
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_db_test.py

In the second test, we used the `database_test` instance to perform a request and check that the object was inserted correctly. This shows the benefit of using asynchronous tests: we can use the same libraries and tools inside our tests. That's all you need to know about `dependency_overrides`.

This feature is also very helpful when you need to write tests for logic involving external services, such as external APIs. Instead of making real requests to those external services during your tests, which could cause issues or incur costs, you'll be able to replace them with another dependency that fakes the requests. To understand this, we've built another example application with an endpoint for retrieving data from an external API:

chapter9_app_external_api.py

```
from typing import Any, Dict

import httpx
from fastapi import FastAPI, Depends

app = FastAPI()

class ExternalAPI:
    def __init__(self) -> None:
        self.client = httpx.AsyncClient(
            base_url="https://dummy.restapiexample.com/api/v1/"
        )

    async def __call__(self) -> Dict[str, Any]:
        async with self.client as client:
            response = await client.get("employees")
            return response.json()

external_api = ExternalAPI()

@app.get("/employees")
async def external_employees(employees: Dict[str, Any] =
    Depends(external_api)):
    return employees
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_external_api.py

To call our external API, we've built a class dependency, as we saw in the *Creating and using a parameterized dependency with a class* section of *Chapter 5, Dependency Injections in FastAPI*. We use HTTPX as an HTTP client to make a request to the external API and retrieve the data. This external API is a dummy API containing fake data, very useful for experiments like this: `https://dummy.restapiexample.com/`.

The `/employees` endpoint is simply injected with this dependency and directly returns the data provided by the external API.

Of course, to test this endpoint, we don't want to make real requests to the external API: it may take time and could be subject to rate limiting. Besides, you may want to test behavior that is not easy to reproduce in the real API, such as errors.

Thanks to `dependency_overrides`, it's very easy to replace our `ExternalAPI` dependency class with another one that returns static data. In the following example, you can see how we implemented such a test:

chapter9_app_external_api_test.py

```
class MockExternalAPI:
    mock_data = {
        "data": [
            {
                "employee_age": 61,
                "employee_name": "Tiger Nixon",
                "employee_salary": 320800,
                "id": 1,
                "profile_image": "",
            }
        ],
        "status": "success",
        "message": "Success",
    }

    async def __call__(self) -> Dict[str, Any]:
        return MockExternalAPI.mock_data

@pytest.fixture(scope="session")
def event_loop():
```

```
    loop = asyncio.get_event_loop()
    yield loop
    loop.close()

@pytest.fixture
async def test_client():
    app.dependency_overrides[external_api] = MockExternalAPI()
    async with LifespanManager(app):
        async with httpx.AsyncClient(app=app, base_url="http://
app.io") as test_client:
            yield test_client

@pytest.mark.asyncio
async def test_get_employees(test_client: httpx.AsyncClient):
    response = await test_client.get("/employees")

    assert response.status_code == status.HTTP_200_OK

    json = response.json()
    assert json == MockExternalAPI.mock_data
    return response.json()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_app_external_api_test.py

Here, you can see that we wrote a simple class called `MockExternalAPI` that returns hardcoded data. All we have to do then is override the original dependency with this one: during the tests, the external API won't be called; we'll only work with the static data.

With the guidelines we've seen so far, you can now write tests for any HTTP endpoints in your FastAPI app. However, there is another kind of endpoint that behaves differently: WebSockets. As we'll see in the next section, unit testing WebSockets is also quite different from what we described for REST endpoints.

Writing tests for WebSocket endpoints

In *Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI*, we explained how WebSockets work and how you can implement such endpoints in FastAPI. As you may have guessed, writing unit tests for WebSockets endpoints is quite different from what we've seen so far.

Unfortunately, we won't be able to reuse HTTPX since, at the time of writing, this client can't communicate with WebSockets. For the time being, our best bet is to use the default `TestClient` provided by Starlette.

To show you this, we'll consider the following WebSocket example:

chapter9_websocket.py

```
from fastapi import FastAPI, WebSocket
from starlette.websockets import WebSocketDisconnect

app = FastAPI()

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    try:
        while True:
            data = await websocket.receive_text()
            await websocket.send_text(f"Message text was:
{data}")
    except WebSocketDisconnect:
        await websocket.close()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_websocket.py

You may have recognized this "echo" example from *Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI*. To test this endpoint, we'll create a new fixture that will instantiate a test client for this application. You can review its implementation in the following example:

chapter9_websocket_test.py

```
import asyncio

import pytest
from fastapi.testclient import TestClient

from chapter9.chapter9_websocket import app

@pytest.fixture(scope="session")
def event_loop():
    loop = asyncio.get_event_loop()
    yield loop
    loop.close()

@pytest.fixture
def websocket_client():
    with TestClient(app) as websocket_client:
        yield websocket_client
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_websocket_test.py

As you can see, we once again took care of defining the `event_loop` fixture, as we explained in the *Setting up testing tools for FastAPI with HTTPX* section.

Then, we implemented the `websocket_client` fixture. The `TestClient` class behaves as a context manager and simply expects the FastAPI application to test the argument. Since we opened a context manager, we once again *yielded* the value to ensure the exit logic is executed after the test. Notice that we don't have to manually take care of the lifespan events, contrary to what we did in the previous sections: `TestClient` is designed to trigger them on its own.

Now, let's write a test for our WebSocket using this fixture:

chapter9_websocket_test.py

```
@pytest.mark.asyncio
async def test_websocket_echo(websocket_client: TestClient):
    with websocket_client.websocket_connect("/ws") as
        websocket:
            websocket.send_text("Hello")

            message = websocket.receive_text()
            assert message == "Message text was: Hello"
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter9/chapter9_websocket_test.py

The first thing to notice is that we still define our test as `async`, with the associated `asyncio` marker, even if `TestClient` works synchronously. Once again, this is useful if you need to call asynchronous services during your tests and limit the issues you may encounter with event loops.

As you can see, the test client exposes a `websocket_connect` method to open a connection to a WebSocket endpoint. It also works as a context manager, giving you the `websocket` variable. It's an object that exposes several methods to either send or receive data. Each of those methods will block until a message has been sent or received.

Here, to test our "echo" server, we send a message thanks to the `send_text` method. Then, we retrieve a message with `receive_text` and assert that it corresponds to what we expect. Equivalent methods also exist for sending and receiving JSON data directly: `send_json` and `receive_json`.

This is what makes WebSocket testing a bit special: you have to think about the sequence of sent and received messages and implement them programmatically to test the behavior of your WebSocket.

Other than that, all the things we've seen so far regarding testing are applicable, especially `dependency_overrides`, when you'll need to use a test database.

Summary

Congratulations! You are now ready to build high-quality FastAPI applications that have been well tested. In this chapter, you learned how to use `pytest`, a powerful and efficient testing framework for Python. Thanks to `pytest` fixtures, you saw how to create a reusable test client for your FastAPI application that can work asynchronously. Using this client, you learned how to make HTTP requests to assert the behavior of your REST API. Finally, we reviewed how to test WebSocket endpoints, which involves a fairly different way of thinking.

Now that you can build a reliable and efficient FastAPI application, it's time to bring it to the whole world! In the next chapter, we'll review the best practices and patterns for preparing a FastAPI application for the world before studying several deployment methods.

10

Deploying a FastAPI Project

Building a good application is great, but it's even better if customers can enjoy it. In this chapter, you'll look at different techniques and the best practices for deploying your FastAPI application to make it available on the web. First, you'll learn how to structure your project to make it ready for deployment by using environment variables to set the configuration options you need, as well as by managing your dependencies properly with `pip`. Once done, we'll show you three ways to deploy your application: with a serverless cloud platform, with a Docker container, and with a traditional Linux server.

In this chapter, we're going to cover the following main topics:

- Setting and using environment variables
- Managing Python dependencies
- Deploying a FastAPI application on a serverless platform
- Deploying a FastAPI application with Docker
- Deploying a FastAPI application on a traditional server

Technical requirements

For this chapter, you'll need a Python virtual environment, similar to the one we set up in *Chapter 1, Python Development Environment Setup*.

You can find all the code examples for this chapter in its dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter10>.

Setting and using environment variables

Before deep diving into the different deployment techniques, we need to structure our application to enable reliable, fast, and secure deployments. One of the key things in this process is handling configuration variables: a database URL, an external API token, a debug flag, and so on. When handling those variables, it's necessary to handle them dynamically instead of hardcoding them in your source code. Why?

First of all, those variables will likely be different in your local environment and in production. Typically, your database URL will point to a local database on your computer while developing but will point to a proper production database in production. This is even more true if you want to have other environments such as a staging or pre-production environment. Furthermore, if we need to change one of the values, we'll have to change the code, commit it, and deploy it again. Thus, we need a convenient mechanism to set those values.

Secondly, it's unsafe to write those values in your code. Values such as database connection strings or API tokens are extremely sensitive. If they appear in your code, they'll likely be committed into your repository: they can be read by anyone who has access to your repository, which causes obvious security issues.

To solve this, we usually use **environment variables**. Environment variables are values that aren't set in the program itself but on the whole system. Most programming languages have the required functions to read those variables from the system. You can try this very easily in a Unix command line:

```
$ export MY_ENVIRONMENT_VARIABLE="Hello" # Set a temporary
variable on the system
$ python
>>> import os
>>> os.getenv("MY_ENVIRONMENT_VARIABLE") # Get it in Python
'Hello'
```

In the Python source code, we can get the value dynamically from the system. During deployment, we'll only have to make sure that we set the correct environment variables on the server. This way, we can easily change a value without redeploying the code and have several deployments of our application containing different configurations sharing the same source code. However, bear in mind that sensitive values that have been set in environment variables could still leak if you don't pay attention; for example, in log files or error stack traces.

To help us with this task, we'll use a very convenient feature of Pydantic: settings management. This allows us to structure and use our configuration variables as we do for any other data model. It even takes care of automatically retrieving the values from environment variables!

For the rest of this chapter, we'll work with an application you can find in `chapter10/project` of our example repository. It's a simple FastAPI application that uses Tortoise ORM, very similar to the one we reviewed in the *Communicating with a SQL database with the Tortoise ORM* section of *Chapter 6, Databases and Asynchronous ORMs*.

Running the commands from the project directory

If you cloned the examples repository, be sure to run the commands shown in this chapter from the `project` directory. In a command line, simply type `cd chapter10/project`.

To structure a settings model, all you need to do is create a class that inherits from `pydantic.BaseSettings`. The following example shows a configuration class with a debug flag, an environment name, and a database URL:

settings.py

```
from pydantic import BaseSettings

class Settings(BaseSettings):
    debug: bool = False
    environment: str
    database_url: str
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter10/project/app/settings.py>

As you can see, creating this class is very similar to creating a standard Pydantic model. We can even define default values, as we did for `debug` here. The good thing with this model is that it works just like any other Pydantic model: it automatically parses the values it finds in environment variables and raises an error if one value is missing in your environment. This way, you can ensure you don't forget any values directly when the app starts.

To use it, we only have to create an instance of this class, as shown in the following code extract:

app.py

```
from app.settings import Settings

settings = Settings()
app = FastAPI()
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter10/project/app/app.py>

Then, you can use it whenever you need one of the variables. In this application, we added a startup event handler that prints all the settings when `debug` is `True`. Besides this, the Tortoise database URL has been set thanks to the `settings` object. You can see this in the following example:

app.py

```
@app.on_event("startup")
async def startup():
    if settings.debug:
        print(settings)

TORTOISE_ORM = {
    "connections": {"default": settings.database_url},
    "apps": {
        "models": {
            "models": ["chapter10.project.models"],
            "default_connection": "default",
        },
    },
},
```

```

}

register_tortoise(
    app,
    config=TORTOISE_ORM,
    generate_schemas=True,
    add_exception_handlers=True,
)

```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter10/project/app/app.py>

You can use settings like any other object in your Python code. If you run this application, you'll likely get the following kind of output:

```

$ uvicorn app.app:app
pydantic.error_wrappers.ValidationError: 2 validation errors
for Settings
environment
  field required (type=value_error.missing)
database_url
  field required (type=value_error.missing)

```

As we mentioned previously, if one value is missing in your environment, Pydantic will raise an error and the application won't start. Let's set those variables and try again:

```

$ export DEBUG="true" ENVIRONMENT="development" DATABASE_
URL="sqlite://chapter10_project.db"
$ uvicorn app.app:app
INFO:      Started server process [1572]
INFO:      Waiting for application startup.
debug=True environment='development' database_url='sqlite://
chapter10_project.db'
INFO:      Application startup complete.
INFO:uvicorn.error:Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press
CTRL+C to quit)
INFO:uvicorn.error:Uvicorn running on http://127.0.0.1:8000
(Press CTRL+C to quit)

```

The application started! You can even see that our startup event handler printed our settings values. Notice that Pydantic is case-insensitive (by default) when retrieving environment variables. By convention, environment variables are usually set in all caps on the system.

Using a .env file

In local development, it's a bit annoying to set environment variables by hand, especially if you're working on several projects at the same time on your machine. To solve this, Pydantic allows you to read the values from a `.env` file. This file contains a simple list of environment variables and their associated values. It's usually easier to edit and manipulate during development.

To make this work, we'll need a new library, `python-dotenv`, whose task is to parse those `.env` files. You can install it as usual with the following command:

```
$ pip install python-dotenv
```

Then, you can edit your `Settings` class, like this:

```
class Settings(BaseSettings):
    debug: bool = False
    environment: str
    database_url: str

    class Config:
        env_file = ".env"
```

You simply have to add a `Config` class and set the `env_file` property to the path of your `.env` file.

Finally, you can create your `.env` file at the root of the project with the following content:

```
DEBUG=true
ENVIRONMENT=development
DATABASE_URL=sqlite://chapter10_project.db
```

And that's it! `settings` will now be read from this `.env` file. If the file is missing, `Settings` will try to read them from the environment variables as usual. Of course, this is only for convenience while developing: this file *shouldn't be committed* and you should rely on *properly set environment variables in production*. To ensure you don't commit this file by accident, it's usually recommended that you add it to your `.gitignore` file.

Creating hidden files such as `.env` files

In Unix systems, files starting with a dot, such as `.env`, are considered hidden files. If you try to create them from the operating system's file explorer, it might show you warnings or even prevent you from doing so. Thus, it's usually more convenient to create them from your IDE, such as Visual Studio Code, or from the command line by using the `touch .env` command, for example.

Great! Our application now supports dynamic configuration variables, which are now easy to set and change on our deployment platforms. Another important thing to take care of is dependencies: we've installed quite a lot of them at this point, but we must make sure they are installed properly during deployments!

Managing Python dependencies

Throughout this book, we've installed libraries using `pip` to add some useful features to our application: FastAPI, of course, but also SQLAlchemy, Tortoise ORM, Pytest, and so on. When deploying a project to a new environment, such as a production server, we have to make sure all those dependencies are installed for our application to work properly. This is also true if you have colleagues that also need to work on the project: they need to know the dependencies they must install on their machines.

Fortunately, `pip` comes with a solution for this so that we don't have to remember all this in our heads. Indeed, most Python projects define a `requirements.txt` file, which contains a list of all Python dependencies. It usually lives at the root of your project. `pip` has a special option for reading this file and installing all the needed dependencies.

When you already have a working environment, such as the one we've used since the beginning of this book, people usually recommend that you run the following command:

```
$ pip freeze
aerich==0.5.3
aiofiles==0.7.0
aiosqlite==0.16.1
alembic==1.6.3
appdirs==1.4.4
asgi-lifespan==1.0.1
asgiref==3.3.4
async-asgi-testclient==1.4.6
asyncio-redis==0.16.0
...
```

The result of `pip freeze` is a list of *every Python package currently installed in your environment*, along with their corresponding versions. This list can be directly used in the `requirements.txt` file.

The problem with this approach is that it lists every package, including the sub-dependencies of the libraries you install. Said another way, in this list, you'll see packages that you don't directly use but that are needed by the ones you installed. If, for some reason, you decide to not use a library anymore, you'll be able to remove it, but it'll be very hard to guess which sub-dependencies it has installed. In the long term, your `requirements.txt` file will grow larger and larger, with lots of dependencies that are useless in your project.

To solve this, some people recommend that you *manually maintain your requirements.txt file*. With this approach, you have to list yourself all the libraries you use, along with their respective versions. During installation, `pip` will take care of installing the sub-dependencies, but they'll never appear in `requirements.txt`. This way, when you remove one of your dependencies, you make sure any useless packages are not kept.

In the following example, you can see the `requirements.txt` file for the project we are working on in this chapter:

requirements.txt

```
fastapi==0.65.2
tortoise-orm[asyncpg]==0.17.4
uvicorn[standard]==0.14.0
gunicorn==20.1.0
```

```
https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter10/project/requirements.txt
```

As you can see, the list is much shorter! Now, whenever we install a new dependency, *our responsibility is to add it manually to requirements.txt*.

A word on alternate package managers such as Poetry, Pipenv and Conda

While exploring the Python community, you may hear about alternate package managers such as Poetry, Pipenv, and Conda. These managers were created to solve some issues posed by `pip`, especially around sub-dependencies management. While they are very good tools, lots of cloud platforms expect a traditional `requirements.txt` file to specify the dependencies, rather than those more modern tools. Therefore, they may not be the best choice for a FastAPI application.

The `requirements.txt` file should be committed along with your source code. When you need to install the dependencies on a new computer or server, you'll simply need to run this command:

```
$ pip install -r requirements.txt
```

Of course, make sure that you're working on proper virtual environments when doing this, as we described in *Chapter 1, Python Development Environment Setup*.

You have probably noticed the `gunicorn` dependency in `requirements.txt`. Let's look at what it is and why it's needed.

Adding Gunicorn as a server process for deployment

In *Chapter 2, Python Programming Specificities*, we briefly introduced WSGI and ASGI protocols. They define the norm and data structure for building web servers in Python. Traditional Python web frameworks, such as Django and Flask, rely on the WSGI protocol. ASGI appeared recently and is presented as the "spiritual successor" of WSGI, providing a protocol for developing web servers running asynchronously. This protocol is at the heart of FastAPI and Starlette.

As we mentioned in *Chapter 3, Developing RESTful APIs with FastAPI*, we use *Uvicorn* to run our FastAPI applications: its role is to accept HTTP requests, transform them according to the ASGI protocol, and pass them to the FastAPI application, which returns an ASGI-compliant response object. Then, *Uvicorn* can form a proper HTTP response from this object.

In the WSGI world, the most widely used server is *Gunicorn*. It has the same role in the context of a Django or Flask application. Why are we talking about it, then? *Gunicorn* has lots of refinements and features that make it more robust and reliable in production than *Uvicorn*. However, *Gunicorn* is designed to work for WSGI applications. So, what can we do?

Actually, we can use both: *Gunicorn* will be used as a robust process manager for our production server. However, we'll specify a special worker class provided by *Uvicorn*, which will allow us to run ASGI applications such as FastAPI. This is the recommended way of doing deployments in the official *Uvicorn* documentation: <https://www.uvicorn.org/deployment/#using-a-process-manager>.

So, let's install Gunicorn to our dependencies by using the following command (remember to add it to your `requirements.txt` file):

```
$ pip install gunicorn
```

If you wish, you can try to run our FastAPI project using Gunicorn by using the following command:

```
$ gunicorn -w 4 -k uvicorn.workers.UvicornWorker app.app:app
```

Its usage is quite similar to Uvicorn, except that we tell it to use a Uvicorn worker. Once again, this is necessary to make it work with an ASGI application. Also, notice the `-w` option. It allows us to set the number of workers to launch for our server. Here, we launch four instances of our application. Then, Gunicorn takes care of load balancing the incoming requests between each worker. This is what makes Gunicorn more robust: if, for any reason, your application blocks the event loop with a synchronous operation, other workers will be able to process other requests while this is happening.

Now, we are ready to deploy our FastAPI application! In the next section, you'll learn how to deploy one on a serverless platform.

Deploying a FastAPI application on a serverless platform

In recent years, serverless platforms have gained a lot of popularity and have become a very common way to deploy web applications. Those platforms completely hide the complexity of setting up and managing a server, giving you the tools to automatically build and deploy your application in minutes. Google App Engine, Heroku, and Azure App Service are among the most popular. Even though they have their own specificities, all these serverless platforms work on the same principles. This is why, in this section, we'll outline the common steps you should follow.

Usually, serverless platforms expect you to provide the source code in the form of a GitHub repository, which you push directly to their servers or that they pull automatically from GitHub. Here, we'll assume that you have a GitHub repository with the source code structured like so:

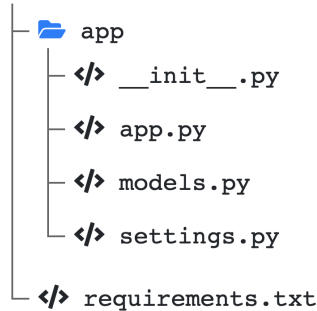


Figure 10.1 – Project structure for serverless deployment

1. Create an account on a cloud platform of your choice. You must do this before you can start any work. It's worth noting that most cloud platforms offer free credits when you are getting started so that you can try their services for free.
2. Install the necessary command-line tools. Most cloud providers supply a complete CLI for managing their services. Typically this is required for deploying your application. Here are the relevant documentation pages for the most popular cloud providers:
 - Google Cloud: <https://cloud.google.com/sdk/gcloud>
 - Microsoft Azure: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>
 - Heroku: <https://devcenter.heroku.com/articles/heroku-cli>
3. Set up the application configuration. Depending on the platform, you'll either have to create a configuration file or use the CLI or the web interface to do this. Here are the relevant documentation pages for the most popular cloud providers:
 - Google App Engine (configuration file): <https://cloud.google.com/appengine/docs/standard/python3/configuring-your-app-with-app-yaml>
 - Azure App Service (web interface and CLI): <https://docs.microsoft.com/en-us/azure/app-service/quickstart-python> and <https://docs.microsoft.com/en-us/azure/app-service/configure-language-python>
 - Heroku (configuration file): <https://devcenter.heroku.com/articles/getting-started-with-python#define-a-procfile>

The key point in this step is to correctly *set the startup command*. As we saw in the previous section, it's essential to set the Uvicorn worker class using the Gunicorn command, as well as set the correct path to your application.

4. Set the environment variables. Depending on the cloud provider, you should be able to do so during configuration or deployment. Remember that they are key for your application to work. Here are the relevant documentation pages for the most popular cloud providers:
 - Google App Engine (configuration file): <https://cloud.google.com/appengine/docs/standard/python/config/appref>
 - Azure App Service (web interface): <https://docs.microsoft.com/en-us/azure/app-service/configure-common#configure-app-settings>
 - Heroku (CLI or web interface): <https://devcenter.heroku.com/articles/config-vars>
5. Deploy the application. Some platforms can automatically deploy when they detect changes on a hosted repository, such as GitHub. Others require that you start a deployment from the command-line tools. Here are the relevant documentation pages for the most popular cloud providers:
 - Google App Engine (CLI): https://cloud.google.com/appengine/docs/standard/python3/testing-and-deploying-your-app#deploying_your_application
 - Azure App Service (continuous deployment or manual Git deployment): <https://docs.microsoft.com/en-us/azure/app-service/deploy-continuous-deployment?tabs=github> and <https://docs.microsoft.com/en-us/azure/app-service/deploy-local-git?tabs=cli>
 - Heroku (CLI): <https://devcenter.heroku.com/articles/getting-started-with-python#deploy-the-app>

Your application should now be live on the platform. Most cloud platforms actually automatically build and deploy Docker containers while following the configuration you provide.

They will make your application available on generic subdomain such as `myapplication.herokuapp.com`. Of course, they also provide mechanisms for binding it to your own domain or subdomain. Here are the relevant documentation pages for the most popular cloud providers:

- Google App Engine: <https://cloud.google.com/appengine/docs/standard/python3/mapping-custom-domains>
- Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/manage-custom-dns-migrate-domain>
- Heroku: <https://devcenter.heroku.com/articles/custom-domains>

Adding database servers

Most of the time, your application will be backed by a database engine, such as PostgreSQL. Fortunately, cloud providers propose fully managed databases, billed according to the computing power, memory, and storage you need. Once created, you'll have access to a connection string to connect to the database instance. All you have to do then is set it in the environment variables of your application. Here are the relevant documentation pages for getting started with managed databases with the most popular cloud providers:

- Google Cloud SQL: <https://cloud.google.com/sql/docs/postgres/create-instance>
- Azure Database for PostgreSQL: <https://docs.microsoft.com/en-us/azure/postgresql/quickstart-create-server-database-portal>
- Amazon RDS: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_GettingStarted.html
- Heroku Postgres: <https://devcenter.heroku.com/articles/heroku-postgresql>

Managing Database Migrations

In *Chapter 6, Databases and Asynchronous ORMs*, we presented you with some tools you can use to manage database migrations: Alembic and Aeric. When working with a cloud database, we advise you to still run them from your local machine so that you have full control of how they are executed, as well as to check if everything goes well. Just be sure to set the correct database connection string.

As we've seen, serverless platforms are the quickest and easiest way to deploy a FastAPI application. However, in some situations, you may wish to have more control of how things are deployed, or you may need system packages that are not available on serverless platforms. In those cases, it may be worthwhile using a Docker container.

Deploying a FastAPI application with Docker

Docker is a widely used technology for containerization. **Containers** are small, self-contained systems running on a computer. Each container contains all the files and configurations necessary for running a single application: a web server, a database engine, a data processing application, and so on. The main goal is to be able to run those applications without worrying about dependency and version conflicts that often happen when trying to install and configure them on the system.

Besides, Docker containers are designed to be *portable and reproducible*: to create a Docker container, you simply have to write a **Dockerfile** containing all the necessary instructions to build the small system, along with all the files and configuration you need. Those instructions are executed during a **build**, which results in a Docker **image**. This image is a package containing your small system, ready to use, that you can easily share on the internet through **registries**. Any developer who has a working Docker installation can then download this image and run it on their system in a container.

Docker has been quickly adopted by developers as it greatly eases the setup of complex development environments, allowing them to have several projects with different system package versions, all without worrying about their installation on their local machine.

However, Docker is not only for local development: it's also widely used for deploying applications to production. Since the builds are reproducible, we can ensure that the environments in local and in production remain the same; which limits issues when passing to production.

In this section, we'll learn how to write a Dockerfile for a FastAPI application, how to build an image, and how to deploy it on a cloud platform.

Writing a Dockerfile

As we mentioned in the introduction to this section a Dockerfile is a set of instructions for building your Docker image, a self-contained system containing all the required components to run your applications. To begin with, all Dockerfiles derive from a base image; usually, this is a standard Linux installation, such as Debian or Ubuntu. From this base, we can copy files from our local machine into the image (usually, the source code of our application) and execute Unix commands; for example, to install packages or execute scripts.

In our case, the creator of FastAPI has created a base Docker image that contains all the necessary tools to run a FastAPI app! All we have to do is start from this image, copy our source files, and install our dependencies! Let's learn how to do that!

First of all, you'll need a working Docker installation on your machine. Follow the official getting started tutorial, which should guide you in this process: <https://docs.docker.com/get-started/>.

To create a Docker image, we simply have to create a file named `Dockerfile` at the root of our project. The following example shows the content of this file for our current project:

Dockerfile

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
ENV APP_MODULE app.app:app
COPY requirements.txt /app
RUN pip install --upgrade pip && \
    pip install -r /app/requirements.txt
COPY ./ /app
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter10/project/Dockerfile>

Let's go through each instruction. The first instruction, `FROM`, is the base image we derive from. Here, we took the `uvicorn-gunicorn-fastapi` image, which was created by the creator of FastAPI. Docker images have tags, which can be used to pick a specific version of the image. Here, we chose Python version 3.7. Lots of variations exist for this image, including ones with newer versions of Python. You can check them out in the official README: <https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker>.

Then, we set the `APP_MODULE` environment variable thanks to the `ENV` instruction. In a Docker image, environment variables can be set at build time, as we did here, or at runtime. `APP_MODULE` is an environment variable defined by the base image. It should point to the path of your FastAPI application: it's the same argument that we set at the end of Uvicorn and Gunicorn commands to launch the application. You can find the list of all the accepted environment variables for the base image in the official README.

Next, we have our first `COPY` statement. As you may have guessed, this instruction will copy a file from your local system to the image. Here, we only copied our `requirements.txt` file. We'll explain why shortly. Notice that we copied the file into the `/app` directory of the image; it's the main working directory defined by the base image.

We then have a `RUN` statement. This instruction is used to execute Unix commands. In our case, we ran `pip` to install our dependencies, following the `requirements.txt` file we just copied. This is essential to make sure all our Python dependencies are present.

Finally, we copied the rest of our source code files into the `/app` directory. Now, let's explain why we separately copied `requirements.txt`. The important thing to understand is that Docker images are built using layers: each instruction will create a new layer in the build system. To improve performance, Docker does its best to reuse layers it has already built. Therefore, if it detects no changes from the previous build, it'll reuse the ones it has in memory without rebuilding them.

By copying the `requirements.txt` file alone and installing the Python dependencies before the rest of the source code, we allow Docker to reuse the layer where the dependencies have been installed. If we edit our source code but not `requirements.txt`, the Docker build will only execute the last `COPY` instruction, reusing all the previous layers. Thus, the image is built in a few seconds instead of minutes.

Most of the time, Dockerfiles end with a `CMD` instruction, which should be the command to execute when the container is started. In our case, we would have used the Gunicorn command we saw in the *Adding Gunicorn as a server* section. However, in our case, the base image is already handling this for us.

Building a Docker image

We can now build our Docker image! From the root of your project, just run the following command:

```
$ docker build -t fastapi-app .
```

The dot (`.`) denotes the path of the root context to build your image – in this case, the current directory. The `-t` option is here to tag the image and give it a practical name.

Docker will then perform the build. You'll see that it'll download the base image and sequentially run your instructions. This should take a few minutes. If you run the command again, you'll experience what we explained earlier about layers: if there is no change, layers are reused and the build takes only a few seconds.

Running a Docker image locally

Before deploying it to production, you can try to run your image locally. To do this, run the following command:

```
$ docker run -p 8000:80 -e ENVIRONMENT=production -e DATABASE_URL=sqlite://./app.db fastapi-app
```

Here, we used the run command with the name of the image we just built. There are, of course, a few options here:

- `-p` allows you to publish ports on your local machine. By default, Docker containers are not accessible on your local machine. If you publish ports, they will be available through `localhost`. On the container side, the FastAPI application is executed on port 80. We publish it on port 8000 on our local machine; that is, `8000:80`.
- `-e` is used to set environment variables. As we mentioned in the *Setting and using environment variables* section, we need those variables to configure our application. Docker allows us to set them easily and dynamically at runtime. Notice that we set a simple SQLite database for testing purposes. However, in production, it should point to a proper database.
- You can review the numerous options of this command in the official Docker documentation: <https://docs.docker.com/engine/reference/commandline/run/#options>.

This command will run your application, which will be accessible through `http://localhost:8000`. Docker will show you the logs in the terminal.

Deploying a Docker image

Now that you have a working Docker image, you can deploy it on virtually any machine that runs Docker. This can be your own server or a dedicated platform. Lots of serverless platforms have emerged to help you deploy container images automatically: Google Cloud Run, Amazon Elastic Container Service, and Microsoft Azure Container Instances are just a few.

Usually, what you have to do is upload (**push**, in Docker jargon) your image to a registry. By default, Docker pulls and pushes images from Docker Hub, the official Docker registry, but lots of services and platforms propose their own registries. Usually, using the private cloud registry proposed by the cloud platform is necessary to deploy it on this platform. Here are the relevant documentation pages for getting started with private registries with the most popular cloud providers:

- Google Artifact Registry: <https://cloud.google.com/artifact-registry/docs/docker/quickstart>
- Amazon ECR: <https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-console.html>
- Microsoft Azure Container Registry: <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-docker-cli?tabs=azure-cli>

If you followed the relevant instructions, you should have a private registry for storing Docker images. The instructions probably showed you how to authenticate your local Docker command line with it and how to push your first image. Basically, all you have to do is tag the image you built with the path to your private registry:

```
$ docker tag fastapi-app aws_account_id.dkr.ecr.region.
amazonaws.com/fastapi-app
```

Then, you need to push it to the registry:

```
$ docker push fastapi-app aws_account_id.dkr.ecr.region.
amazonaws.com/fastapi-app
```

Your image is now safely stored in the cloud platform registry. You can now use their serverless container platform to deploy it automatically. Here are the relevant documentation pages for getting started with private registries with the most popular cloud providers:

- Google Cloud Run: <https://cloud.google.com/run/docs/quickstarts/build-and-deploy/python>
- Amazon Elastic Container Service: <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/getting-started-ecs-ec2.html>
- Microsoft Azure Container Instances: <https://docs.microsoft.com/en-us/azure/container-instances/container-instances-tutorial-deploy-app>

Of course, you'll be able to set the environment variables just like you can for fully managed apps. Those environments also provide lots of options for tuning the scalability of your containers, both vertically (using more powerful instances) and horizontally (spawn more instances).

Once done, your application should be live on the web! The great thing about deploying Docker images compared to automated serverless platforms is that you are not limited to the features supported by the platform: you can deploy anything, even complex applications that require a lot of exotic packages, without worrying about compatibility.

At this point, we've seen the easiest and most efficient ways to deploy a FastAPI application. However, you may wish to deploy one the old-fashioned way and manually set up your server. In the next section, we'll provide some guidelines to do so.

Deploying a FastAPI application on a traditional server

In some situations, you may not have the chance to use a serverless platform to deploy your application. Some security or regulatory policies may force you to deploy on physical servers with specific configurations. In this case, it's worth knowing some basic things so that you can deploy your application on traditional servers.

In this section, we'll consider you are working on a Linux server:

1. First of all, make sure a *recent version of Python has been installed* on your server, ideally with the version matching the one you used in development. The easiest way to do this is to set up *pyenv*, as we saw in *Chapter 1, Python Development Environment Setup*.
2. To retrieve your source code and keep it in sync with your latest developments, you can *clone your Git repository* on your server. This way, you only have to pull the changes and restart the server process to deploy a new version.
3. Set up a *Python virtual environment*, as we explained in *Chapter 1, Python Development Environment Setup*. You can install the dependencies with `pip` thanks to your `requirements.txt` file.
4. At that point, you should be able to run Gunicorn and start serving your FastAPI application. However, some improvements are strongly recommended.
5. Use a *process manager* to ensure your Gunicorn process is always running and restarted when the server is restarted. A good option for this is *Supervisor*. The Gunicorn documentation provides good guidelines for this: <https://docs.gunicorn.org/en/stable/deploy.html#supervisor>.

6. It's also recommended to *put Gunicorn behind an HTTP proxy* instead of directly putting it on the front line. Its role is to handle SSL connections, perform load balancing, and serve static files such as images or documents. The Gunicorn documentation recommends using Nginx for this task and provides a basic configuration: <https://docs.gunicorn.org/en/stable/deploy.html#nginx-configuration>.

As you can see, in this context, there's quite a lot of configurations and decisions to make regarding your server configuration. Of course, you should also pay attention to security and make sure your server is well-protected against the usual attacks. In the following DigitalOcean tutorial, you'll find some guidelines for securing your server: <https://www.digitalocean.com/community/tutorials/recommended-security-measures-to-protect-your-servers>.

If you're not an experienced system administrator, we recommend that you favor serverless platforms: professional teams handle security, system updates, and server scalability for you, letting you focus on what matters most for you: developing a great application!

Summary

Your application is now live on the web! In this chapter, we covered the best practices to apply before deploying your application to production: use environment variables to set configuration options, such as database URLs, and manage your Python dependencies with a `requirements.txt` file. Then, we showed you how to deploy your application to a serverless platform, which handles everything for you by retrieving your source code, packaging it with its dependencies, and serving it on the web. Next, you learned how to build a Docker image for FastAPI using the base image created by the creator of FastAPI. As you've seen, it allows you to be flexible while configuring the system, but you can still deploy it in a few minutes with a serverless platform that's compatible with containers. Finally, we provided you with some guidelines for manual deployment on a traditional Linux server.

This marks the end of the second part of this book. You should now be confident in writing efficient, reliable FastAPI applications and be able to deploy them on the web.

In the next chapter, we will begin some data science tasks and integrate them efficiently in a FastAPI project.

Section 3: Build a Data Science API with Python and FastAPI

This section will introduce the most common libraries used in Python to perform data science-related tasks. We'll see how to integrate those tools in a FastAPI backend with performance and maintainability in mind.

This section comprises the following chapters:

- *Chapter 11, Introduction to NumPy and pandas*
- *Chapter 12, Train Machine Learning Models with scikit-learn*
- *Chapter 13, Create an Efficient Prediction API Endpoint with FastAPI*
- *Chapter 14, Implement a Real-Time Face Detection System Using WebSockets with FastAPI and OpenCV*

11

Introduction to NumPy and pandas

In recent years, Python has gained a lot of popularity in the data science field. Its very efficient and readable syntax makes the language a very good choice for scientific research, while still being suitable for production workloads: it's very easy to deploy research projects into real applications that will bring value to users. Thanks to this growing interest, a lot of specialized Python libraries have emerged. The most well known are probably NumPy and pandas. Their goal is to provide a set of tools to manipulate a big set of data in an efficient way, much more than what we could actually achieve with standard Python, and we'll show how and why in this chapter. NumPy and pandas are at the heart of most data science applications in Python; knowing them is therefore the first step on your journey into Python for data science.

In this chapter, we're going to cover the following main topics:

- Getting started with NumPy
- Manipulating arrays with NumPy: computation, aggregations, comparisons
- Getting started with pandas

Technical requirements

You'll need a Python virtual environment, as we set up in *Chapter 1, Python Development Environment Setup*.

You'll find all the code examples of this chapter in the dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter11>.

Getting started with NumPy

In *Chapter 2, Python Programming Specificities*, we stated that Python is a dynamically typed language. This means that the interpreter automatically detects the type of a variable at runtime, and this type can even change throughout the program. For example, you can do something like this in Python:

```
$ python
>>> x = 1
>>> type(x)
<class 'int'>
>>> x = "hello"
>>> type(x)
<class 'str'>
```

The interpreter was able to determine the type of `x` at each assignation.

Under the hood, the standard implementation of Python, CPython, is written in C. The C language is a compiled and statically typed language. This means that the nature of the variables is fixed at compile time, and they can't change during execution. Thus, in the Python implementation, a variable doesn't only consist in its value: it's actually a structure containing information about the variable, including its type and size, in addition to its value.

Thanks to this, we can manipulate variables very dynamically in Python. However, it comes at a cost: each variable has a significantly higher memory footprint to store all its metadata than just the plain value.

This is particularly true for data structures. Say we consider a simple list like this:

```
$ python
>>> l = [1, 2, 3, 4, 5]
```

Each item of the list is a Python integer, with all the metadata associated. In a statically typed language such as C, the same list would only be a suite of values in memory sharing the same type.

Let's now imagine a big set of data, like the kind we usually encounter in data science: the cost of storing it in memory would be huge. That's exactly the purpose of NumPy: provide a powerful and efficient array structure to manipulate a big set of data. Under the hood, it uses a fixed-type array, meaning all elements of the structure are of the same type, which allows NumPy to get rid of the costly metadata of every single element. Moreover, common arithmetic operations, such as additions or multiplications, are much faster. In the *Manipulating arrays with NumPy – computation, aggregations, comparisons* section of this chapter, we'll make a speed comparison to show you the difference with standard Python lists.

To get started, let's install NumPy using the following command:

```
$ pip install numpy
```

In a Python interpreter, we can now import the library:

```
$ python
>>> import numpy as np
```

Notice that, by convention, *NumPy is always imported with the alias np*. Let's now discover its basic features!

Creating arrays

To create an array with NumPy, we can simply use the `array` function and pass it a Python list:

```
>>> np.array([1, 2, 3, 4, 5])
array([1, 2, 3, 4, 5])
```

NumPy will detect the nature of the Python list. However, we can force the resulting type by using the `dtype` argument:

```
>>> np.array([1, 2, 3, 4, 5], dtype=np.float64)
array([1., 2., 3., 4., 5.] )
```

All elements were upcasted to the specified type. It is key to remember that a **NumPy array is of a fixed type**. This means that every element will have the same type and NumPy will silently cast a value to the array type. For example, let's consider an integer list in which we want to insert a floating-point value:

```
>>> l = np.array([1, 2, 3, 4, 5])
>>> l[0] = 13.37
>>> l
array([13,  2,  3,  4,  5])
```

The value 13.37 has been truncated to fit into an integer.

If the value cannot be cast to the type of array, an error is raised. For example, let's try to change the first element by using a string:

```
>>> l[0] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

As we said in the introduction to this section, Python lists are not very efficient for large datasets. This is why it's generally more efficient to use NumPy functions to create arrays. The most commonly used ones are generally the following:

- `np.zeros`, to create an array filled with zeros
- `np.ones`, to create an array filled with ones
- `np.empty`, to create an empty array of the desired size in memory, without initializing the values
- `np.arange`, to create an array with a range of elements

Let's see them in action:

```
>>> np.zeros(5)
array([0., 0., 0., 0., 0.])
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
>>> np.empty(5)
array([1., 1., 1., 1., 1.])
>>> np.arange(5)
array([0, 1, 2, 3, 4])
```

Notice that the result of `np.empty` can vary: since the values in the array are not initialized, **they take whatever value there is currently in this memory block**. The main motivation behind this function is speed, allowing you to quickly allocate memory; but don't forget to fill every element after.

By default, NumPy create arrays with a floating-point type (`float64`). Once again, by using the `dtype` argument, you can force another type to be used:

```
>>> np.ones(5, dtype=np.int32)
array([1, 1, 1, 1, 1], dtype=int32)
```

NumPy provides a wide range of types, allowing you to finely optimize the memory consumption of your program by selecting the right type for your data. You can find the whole list of types supported by NumPy in the official documentation: <https://numpy.org/doc/stable/reference/arrays.scalars.html#sized-aliases>.

NumPy also proposes a function to create an array with random values:

```
>>> np.random.seed(0) # Set the random seed to make examples
reproducible
>>> np.random.randint(10, size=5)
array([5, 0, 3, 3, 7])
```

The first argument is the maximum range of the random value, and the `size` argument sets the number of values to generate.

Until now, we showed how to create one-dimensional arrays. However, the great strength of NumPy is that it natively handles multi-dimensional arrays! For example, let's create a 3 x 4 matrix:

```
>>> m = np.ones((3,4))
>>> m
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```


NumPy did create an array with three rows and four columns! All we had to do was to pass a tuple to the NumPy function to specify our dimensions. When having such an array, NumPy gives us access to properties to know the number of dimensions, as well as the shape and size of it:

```
>>> m.ndim
2
>>> m.shape
(3, 4)
>>> m.size
12
```

Accessing elements and sub-arrays

NumPy arrays closely follow the standard Python syntax to manipulate lists. Therefore, to access an element in a one-dimensional array, just do the following:

```
>>> l = np.arange(5)
>>> l[2]
2
```

For multi-dimensional arrays, we just have to add another index:

```
>>> np.random.seed(0)
>>> m = np.random.randint(10, size=(3,4))
>>> m
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
>>> m[1][2]
3
```

Of course, this can be used to re-assign elements:

```
>>> m[1][2] = 42
>>> m
array([[ 5,  0,  3,  3],
       [ 7,  9, 42,  5],
       [ 2,  4,  7,  6]])
```

But that's not all. Thanks to the slicing syntax, we can access sub-arrays with a start and end index and even a step. For example, on a one-dimensional array, we can do the following:

```
>>> l = np.arange(5)
>>> l
array([0, 1, 2, 3, 4])
>>> l[1:4] # From index 1 (inclusive) to 4 (exclusive)
array([1, 2, 3])
>>> l[::2] # Every second element
array([0, 2, 4])
```

This is exactly what we saw for standard Python lists in *Chapter 2, Python Programming Specificities*. Of course, it also works for multi-dimensional arrays, with one slice for each dimension:

```
>>> np.random.seed(0)
>>> m = np.random.randint(10, size=(3,4))
>>> m
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
>>> m[1:, 0:2] # From row 1 to end and column 0 to 2
array([[7, 9],
       [2, 4]])
>>> m[:, 3:] # Every row, only last column
array([[3],
       [5],
       [6]])
```

You can assign those sub-arrays to variables. However, for performance reasons, NumPy doesn't copy the values by default: it's only a **view** (or shallow copy), a representation of the existing data. This is important to bear in mind because if you change a value on the view, it will also change the value on the original array:

```
>>> v = m[:, 3:]
>>> v[0][0] = 42
>>> v
array([[42],
```

```
[ 5],  
 [ 6]])  
>>> m  
array([[ 5,  0,  3, 42],  
       [ 7,  9,  3,  5],  
       [ 2,  4,  7,  6]])
```

If you need to **deep copy** the values, you just have to use the `copy` method on the array:

```
>>> v = m[:, :3].copy()
```

`v` is now a separate copy of `m`, and changes on its values won't change the values in `m`.

You now have the basics of handling arrays with NumPy. As we've seen, the syntax is very similar to standard Python. The key points to remember when working with NumPy are the following:

- NumPy arrays are of fixed types, meaning every item in the array are of the same type.
- NumPy natively handles multi-dimensional arrays and allows us to subset them using the standard slicing notation.

Of course, NumPy can do much more than that: actually, it can apply common computations to those arrays in a very performant way.

Manipulating arrays with NumPy – computation, aggregations, comparisons

As we said, NumPy is all about manipulating large arrays with great performance and controlled memory consumption. Let's say, for example, that we want to compute the double of each element in a large array. In the following example, you can see an implementation of such a function with a standard Python loop:

chapter11_compare_operations.py

```
import numpy as np  
  
np.random.seed(0) # Set the random seed to make examples  
reproducible
```

```
m = np.random.randint(10, size=1000000) # An array with a
million of elements
```

```
def standard_double(array):
    output = np.empty(array.size)
    for i in range(array.size):
        output[i] = array[i] * 2
    return output
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter11/chapter11_compare_operations.py

We instantiate an array with a million random integers. Then, we have our function building an array with the double of each element. Basically, we first instantiate an empty array of the same size before looping over each element to set the double.

Let's measure the performance of this function. In Python, there is a standard module, `timeit`, dedicated to this purpose. We can use it directly from the command line and pass in argument-valid Python statements that we want to measure performance. The following command will measure the performance of `standard_double` with our big array:

```
$ python -m timeit "from chapter11.chapter11_compare_operations
import m, standard_double; standard_double(m)"
1 loop, best of 5: 315 msec per loop
```

The results will vary depending on your machine, but the magnitude should be equivalent. What `timeit` does is to repeat your code a certain number of times and measure its execution time. Here, our function took around 300 milliseconds to compute the double of each element in our array. For such simple computations on a modern computer, that's not very impressive.

Let's compare this with the equivalent operation using NumPy syntax. You can see it in the next sample:

chapter11_compare_operations.py

```
def numpy_double(array):  
    return array * 2
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter11/chapter11_compare_operations.py

The code is much shorter! NumPy implements the basic arithmetic operations and can apply them to each element of the array. By multiplying the array by a value directly, we implicitly tell NumPy to multiply each element by this value. Let's measure the performance with `timeit`:

```
$ python -m timeit "from chapter11.chapter11_compare_operations  
import m, numpy_double; numpy_double(m) "  
500 loops, best of 5: 667 usec per loop
```

Here, the best loop achieved the computation in 600 microseconds! That's almost a thousand times faster than the previous function! How can we explain such a variation? In a standard loop, Python, because of its dynamic nature, has to check for the type of value at each iteration to apply the right function for this type, which adds significant overhead. With NumPy, the operation is deferred to an optimized and compiled loop where types are known ahead of time, which saves a lot of useless checks.

We once again see here the benefits of NumPy arrays over standard lists when working on a large dataset: it implements operations natively to help you make computations very fast.

Adding and multiplying arrays

As you saw in the previous example, NumPy supports the arithmetic operators to make operations over arrays.

This means that you can operate directly over two arrays of the same dimensions:

```
>>> np.array([1, 2, 3]) + np.array([4, 5, 6])  
array([5, 7, 9])
```

In this case, NumPy applies the operation element-wise. But it also works in certain situations if one of the operands is not of the same shape:

```
>>> np.array([1, 2, 3]) * 2
array([2, 4, 6])
```

NumPy automatically understands that it should multiply each element by two. This is called **broadcasting**: NumPy "expands" the smaller array to match the shape of the larger array. The previous example is equivalent to this one:

```
>>> np.array([1, 2, 3]) * np.array([2, 2, 2])
array([2, 4, 6])
```

Note that even if those two examples are conceptually equivalent, the first one is more memory-efficient and computationally efficient: NumPy is smart enough to use only one value, "two", without having to create a full array of "two".

More generally, broadcasting works if the rightmost dimensions of the arrays are of the same size or if one of them is one. For example, we can add an array of dimensions 4 x 3 to an array of dimensions 1 x 3:

```
>>> a1 = np.ones((4, 3))
>>> a1
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
>>> a2 = np.ones((1, 3))
>>> a2
array([[1., 1., 1.]])
>>> a1 + a2
array([[2., 2., 2.],
       [2., 2., 2.],
       [2., 2., 2.],
       [2., 2., 2.]])
```

However, adding an array of dimensions 4×3 to an array of dimensions 1×4 is not possible:

```
>>> a3 = np.ones((1, 4))
>>> a3
array([[1., 1., 1., 1.]])
>>> a1 + a3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with
shapes (4,3) (1,4)
```

If this sounds complicated or confusing, that's normal; it takes time to understand it conceptually, especially in three or more dimensions. For a more detailed explanation of the concept, take time to read the related article in the official documentation: <https://numpy.org/doc/stable/user/theory.broadcasting.html>.

Aggregating arrays – sum, min, max, mean...

When working with arrays, we often need to summarize the data to extract some meaningful statistics: the mean, the minimum, the maximum... Fortunately, NumPy also provides those operations natively. Quite simply, they are provided as methods that you can call directly from an array:

```
>>> np.arange(10).mean()
4.5
>>> np.ones((4,4)).sum()
16.0
```

You can find the whole list of aggregating operations in the official documentation: <https://numpy.org/doc/stable/reference/arrays.ndarray.html#calculation>.

By default, those operations will aggregate every value in the array. However, you can apply them per axis for multi-dimensional arrays:

```
>>> m = np.array(
[[6, 5, 1, 1],
 [8, 9, 3, 2],
 [9, 3, 8, 5],
 [1, 0, 1, 9]]
```

```

)
>>> m.sum(axis=0) # Sum on the rows axis (the first dimension)
array([24, 17, 13, 17])
>>> m.sum(axis=1) # Sum on the columns axis (the second
dimension)
array([13, 22, 25, 11])

```

Comparing arrays

NumPy also implements the standard comparison operators to compare arrays. As with arithmetic operators, which we saw in the *Adding and multiplying arrays* section, broadcasting rules apply. This means that you can compare an array with a single value:

```

>>> l = np.array([1, 2, 3, 4])
>>> l < 3
array([ True,  True, False, False])

```

And you can also compare arrays with arrays, given that they are compatible on the basis of the broadcasting rules:

```

>>> m = np.array(
    [[1., 5., 9., 13.],
     [2., 6., 10., 14.],
     [3., 7., 11., 15.],
     [4., 8., 12., 16.]]
)
>>> m <= np.array([1, 5, 9, 13])
array([[ True,  True,  True,  True],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]])

```

The resulting array is filled with the Boolean result of the comparison for each element.

That's it for this very quick introduction to NumPy. There is a lot more to know and discover with this library, so we strongly encourage you to read the official user guide: <https://numpy.org/doc/stable/user/index.html>.

For the rest of this book, this should be enough for you to understand the future examples. Let's now have a look at a library often cited and used alongside NumPy: pandas.

Getting started with pandas

In the previous section, we introduced NumPy and its ability to efficiently store and work with a large array of data. We'll now introduce another widely used library in data science: pandas. This library is built on top of NumPy to provide convenient data structures able to efficiently store large datasets with *labeled rows and columns*. This is, of course, especially handy when working with most datasets representing real-world data that we want to analyze and use in data science projects.

To get started, we will, of course, install the library with the usual command:

```
$ pip install pandas
```

Once done, we can start to use it in a Python interpreter:

```
$ python
>>> import pandas as pd
```

Just like we alias numpy as np, the convention is to alias pandas as pd when importing it.

Using pandas Series for one-dimensional data

The first pandas data structure we'll introduce is `Series`. This data structure behaves very similarly to a one-dimensional array in NumPy. To create one, we can simply initialize it with a list of values:

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Under the hood, pandas create a NumPy array. As such, it uses the same data types to store the data. You can verify this by accessing the `values` property of the `Series` object and check its type:

```
>>> type(s.values)
<class 'numpy.ndarray'>
```

Indexing and slicing work exactly the same way as in NumPy:

```
>>> s[0]
1
>>> s[1:3]
1    2
2    3
dtype: int64
```

So far, this is not very different from a regular NumPy array. As we said, the main purpose of pandas is to *label the data*. To allow this, pandas data structures maintain an index to allow this data labeling. It is accessible through the `index` property:

```
>>> s.index
RangeIndex(start=0, stop=5, step=1)
```

Here, we have a simple range integer index, but we can actually have any arbitrary index. In the next example, we create the same series, labeling each value with a letter:

```
>>> s = pd.Series([1, 2, 3, 4, 5], index=["a", "b", "c", "d", "e"])
>>> s
a    1
b    2
c    3
d    4
e    5
```

The `index` argument on the `Series` initializer allows us to set the list of labels. We can now access values with those labels instead:

```
>>> s["c"]
3
```

Surprisingly, even slicing notation works with those kinds of labels:

```
>>> s["b":"d"]
b    2
c    3
d    4
dtype: int64
```

Under the hood, pandas keep the order of the index to allow such useful notations. Notice, however, that with this notation, the **last index is inclusive** (d is included in the result), unlike standard index notation, where the last index is exclusive:

```
>>> s[1:3]
b    2
c    3
dtype: int64
```

To avoid confusion between those two styles, pandas exposes two special notations to explicitly indicate which indexing style you wish to use: `loc` (label notation with the last index being inclusive) and `iloc` (standard index notation). You can read more about this in the official documentation: https://pandas.pydata.org/docs/user_guide/indexing.html#different-choices-for-indexing.

Series can also be instantiated directly from dictionaries:

```
>>> s = pd.Series({"a": 1, "b": 2, "c": 3, "d": 4, "e": 5})
>>> s
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

In this case, the keys of the dictionaries are used as labels.

Of course, in the real world, you'll more likely have to work with two-dimensional (or more!) datasets. This is exactly what DataFrames are for!

Using pandas DataFrames for multi-dimensional data

Most of the time, datasets consist of two-dimensional data, where you have several columns for each row, as in a classic spreadsheet application. In pandas, DataFrames are designed to work this kind of data. As for `Series`, it can work with a large set of data that is labeled both by rows and columns.

The following examples will use a tiny dataset representing the number of tickets (paid and free) delivered in French museums in 2018. Let's consider we have this data in the form of two dictionaries:

```
>>> paid = {"Louvre Museum": 5988065, "Orsay Museum": 1850092,
            "Pompidou Centre": 2620481, "National Natural History Museum":
            404497}
>>> free = {"Louvre Museum": 4117897, "Orsay Museum": 1436132,
            "Pompidou Centre": 1070337, "National Natural History Museum":
            344572}
```

Each key in those dictionaries is a label for a row. We can build a `DataFrame` directly from those two dictionaries like this:

```
>>> museums = pd.DataFrame({"paid": paid, "free": free})
>>> museums
```

	paid	free
Louvre Museum	5988065	4117897
Orsay Museum	1850092	1436132
Pompidou Centre	2620481	1070337
National Natural History Museum	404497	344572

The `DataFrame` initializer accepts a dictionary of dictionaries, where keys represent the label for the columns.

We can have a look at the `index` property, storing the rows index, and the `columns` property, storing the columns index:

```
>>> museums.index
Index(['Louvre Museum', 'Orsay Museum', 'Pompidou Centre',
      'National Natural History Museum'],
      dtype='object')
>>> museums.columns
Index(['paid', 'free'], dtype='object')
```

Once again, we can now use indexing and slicing notation to get subsets of columns or rows:

```
>>> museums["free"]
Louvre Museum          4117897
Orsay Museum           1436132
Pompidou Centre        1070337
National Natural History Museum  344572
Name: free, dtype: int64
>>> museums["Louvre Museum":"Orsay Museum"]
           paid    free
Louvre Museum  5988065  4117897
Orsay Museum   1850092  1436132
>>> museums["Louvre Museum":"Orsay Museum"]["paid"]
Louvre Museum    5988065
Orsay Museum     1850092
Name: paid, dtype: int64
```

Something that is even more powerful, you can write a Boolean condition inside the brackets to match some data. This operation is called **masking**:

```
>>> museums[museums["paid"] > 2000000]
           paid    free
Louvre Museum  5988065  4117897
Pompidou Centre 2620481  1070337
```

Finally, you can easily set new columns with this very same indexing notation:

```
>>> museums["total"] = museums["paid"] + museums["free"]
>>> museums
           paid    free    total
Louvre Museum  5988065  4117897  10105962
Orsay Museum   1850092  1436132   3286224
Pompidou Centre 2620481  1070337   3690818
National Natural History Museum  404497  344572   749069
```

As you can see, just like NumPy arrays, pandas fully supports arithmetic operations over two series or DataFrames.

Of course, all the basic aggregation operations are supported, including mean and sum:

```
>>> museums["total"].sum()
17832073
>>> museums["total"].mean()
4458018.25
```

You can find the whole list of operations available in the official documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#descriptive-statistics.

Importing and exporting CSV data

One very common way of sharing datasets is through CSV files. This format is very convenient because it only consists of a simple text file, each line representing a row of data, with each column separated by a comma. Our simple museums dataset is available in the examples repository as a CSV file, which you can see in the next sample:

museums.csv

```
name,paid,free
Louvre Museum,5988065,4117897
Orsay Museum,1850092,1436132
Pompidou Centre,2620481,1070337
National Natural History Museum,404497,344572
```

<https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter11/museums.csv>

Importing CSV files is so common that pandas provides a function to load a CSV file into a DataFrame directly:

```
>>> museums = pd.read_csv("./chapter11/museums.csv", index_
col=0)
>>> museums
```

	paid	free
name		
Louvre Museum	5988065	4117897
Orsay Museum	1850092	1436132
Pompidou Centre	2620481	1070337

```
National Natural History Museum    404497    344572
```

The function simply expects the path to the CSV file. Several arguments are available to finely control the operation: here, we used `index_col` to specify the index of the column that should be used as row labels. You can find the whole list of arguments in the official documentation: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html.

Of course, the opposite operation exists to export a DataFrame to a CSV file:

```
>>> museums["total"] = museums["paid"] + museums["free"]
>>> museums.to_csv("museums_with_total.csv")
```

We will conclude this very quick introduction to pandas here. Of course, we've only covered the tip of the iceberg here and we recommend that you go through the official user guide to know more: https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html.

Still, you should now be able to perform basic operations and operate efficiently on large datasets.

Summary

Great! You now have a grasp of the ins and outs of NumPy and pandas. Basically, those libraries are the essential tool for data scientists in Python. By relying on optimized and compiled code, they allow you to load and manipulate large set of data in Python, without sacrificing performance. To allow this, they define fixed-type data structures, meaning each value in the dataset should be of the same type. This is what enables efficient memory consumption and fast computations.

Even though those basics should be enough for you to get started, we recommend that you spend some time on the official user guides and tinker with those a bit to discover all their aspects.

As we said in the introduction, NumPy and pandas are at the heart of most data science applications in Python. In the next chapter, we'll see how they will help us in machine learning tasks, along with the well-known machine learning library scikit-learn.

12

Training Machine Learning Models with scikit-learn

As we mentioned in the introduction of the previous chapter, Python has gained a lot of popularity in the data science field. We've seen that libraries such as NumPy and pandas have emerged to handle big datasets efficiently in Python. Those libraries are the foundation for libraries dedicated to **machine learning (ML)**, such as the famous scikit-learn library, a complete toolset for implementing most of the algorithms and techniques that are used daily by data scientists. In this chapter, we'll provide a quick introduction to ML, what it is about, what it tries to solve, and how. Then, we'll learn how to use scikit-learn to train and test ML models. We'll also have a deeper look at two classical ML models, Naive Bayes models and support vector machines, both of which can perform surprisingly well if used correctly.

In this chapter, we're going to cover the following main topics:

- What is machine learning?
- Basics of scikit-learn
- Classifying data with Naive Bayes models
- Classifying data with support vector machines

Technical requirements

You'll need a Python virtual environment, similar to the one we set up in *Chapter 1, Python Development Environment Setup*.

You can find all the code examples for this chapter in this book's dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter12>.

What is machine learning?

ML is often seen as a sub-field of artificial intelligence. While this categorization is a subject of debate, ML has had a lot of exposure in recent years due to its vast and visible field of applications, such as spam filters, natural language processing, and autonomous driving.

ML is a field where we build mathematical models from existing data so that the machine can understand this data by itself. The machine is "learning" in the sense that the developer doesn't have to program a step-by-step algorithm to solve the problem, which would be impossible for complex tasks. Once a model has been "trained" on existing data, it can be used to predict new data or understand new observations.

Consider the spam filter example: if we have a sufficiently large collection of emails manually labeled "spam" or "not spam," we can use ML techniques to build a model that can tell us if a new incoming email is spam or not.

Before we look at this with scikit-learn, we'll review the most fundamental concepts of ML.

Supervised versus unsupervised learning

ML techniques can be divided into two main categories: **supervised learning** and **unsupervised learning**.

With supervised learning, the existing dataset is already labeled, which means we have both the inputs (the characteristics of an observation), known as **features**, and the outputs. If we consider the spam filter example here, the features could be the frequencies of each word and the **label** could be the category; that is, "spam" or "not-spam".

Supervised learning is subdivided into two groups:

- **Classification problems**, to classify data with a finite set of categories; for example, the spam filter
- **Regression problems**, to predict continuous numerical values; for example, the number of rented electric scooters, given the day of the week, the weather, and the location

Unsupervised learning, on the other hand, operates on data without any reference to a label. The goal here is to discover interesting patterns from the features themselves. The two main problems that unsupervised learning tries to solve are as follows:

- **Clustering**, where we want to find groups of similar data points; for example, a recommender system to suggest products that you might like, given what other people similar to you like.
- **Dimensionality reduction**, where the goal is to find a more compact representation of datasets that contain a lot of different features. Doing this will allow us to keep only the most meaningful and discriminant features while working with smaller dataset dimensions.

Model validation

One of the key aspects of ML is evaluating whether your model is performing well or not. How can you say that your model will perform well on newly observed data? When building your model, how can you tell if one algorithm performs better than another? All of these questions can and should be answered with model validation techniques.

As we mentioned previously, ML methods start with an existing set of data that we'll use to train a model.

Intuitively, we may want to use all the data we have to train our model. Once done, what can we do to test it? We could apply our model to the same data and see if the output is correct... and we would get a surprisingly good result! Here, we are testing the model with the same data we used to train it. Obviously, the model will overperform on this data because it has already seen it. As you may have guessed, this is not a reliable way to measure the accuracy of our model.

The right way to validate a model is to split the data into two: we keep one part for training the data and another for testing it. This is known as the **holdout set**. This way, we'll test the model on data that it has never seen before and compare the result that's predicted by the model with the real value. Hence, the accuracy we are measuring is much more sensible.

This technique works well; however, it poses a problem: by retaining some data, we are losing precious information that could have helped us build a better model. This is especially true if our initial dataset is small. To solve this, we can use **cross-validation**. With this method, we once again split the data into two sets. This time, we are training the model twice, using each set as training and testing sets. You can see a schematic representation of this operation in the following diagram:



Figure 12.1 – Two-fold cross-validation

At the end of the operation, we obtain two accuracies, which will give us a better overview of how our model performs on the whole dataset. This technique can be applied to help us perform more trials with a smaller testing set, as shown in the following diagram:

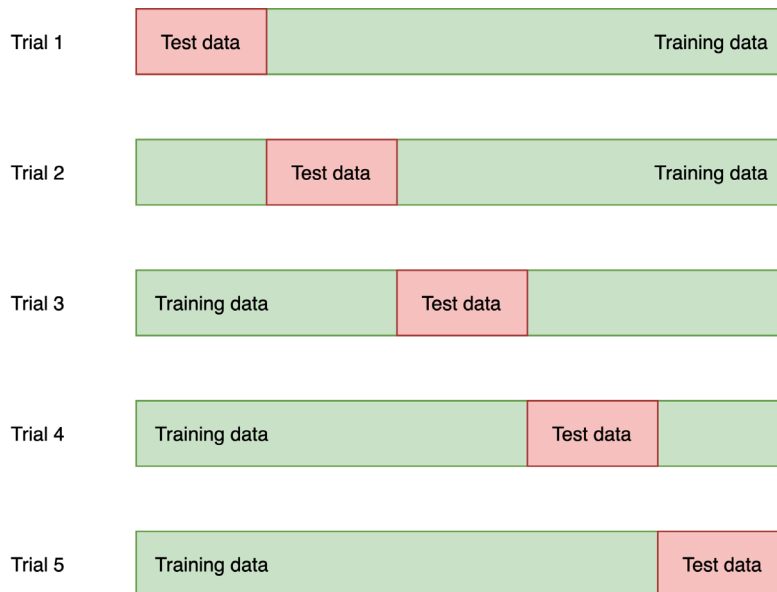


Figure 12.2 – Five-fold cross-validation

We'll stop here regarding this very quick introduction to ML. We've barely scratched the surface: ML is a vast and complex field, and there are lots of books dedicated to this subject. Still, this information should be sufficient to help you understand the basic concepts of scikit-learn, which we'll show throughout the rest of this chapter.

Basics of scikit-learn

Now, let's focus on scikit-learn, an essential ML library for Python. It implements dozens of classic ML models, but also numerous tools to help you while training them, such as pre-processing methods and cross-validation.

The first thing you must do to get started is install it in your Python environment:

```
$ pip install scikit-learn
```

We can now start our scikit-learn journey!

Training models and predicting

In scikit-learn, ML models and algorithms are called **estimators**. Each is a Python class that implements the same methods. In particular, we have `fit`, which is used to train a model, and `predict`, which is used to run the trained model on new data.

To try this, we'll load a sample dataset. scikit-learn comes with a few toy datasets that are very useful for performing experiments. You can find out more about them in the official documentation: <https://scikit-learn.org/stable/datasets.html>.

Here, we'll use the *digits dataset*, a collection of pixels matrices representing handwritten digits. As you may have guessed, the goal of this dataset is to train a model to automatically recognize handwritten digits. The following example shows how to load this dataset:

chapter12_load_digits.py

```
from sklearn.datasets import load_digits

digits = load_digits()

data = digits.data
targets = digits.target

print(data[0].reshape((8, 8))) # First handwritten digit 8 x 8
matrix

print(targets[0]) # Label of first handwritten digit
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_load_digits.py

Notice that the toy dataset's functions are imported from the `datasets` package of scikit-learn. The `load_digits` function returns an object that contains the data and some metadata.

The most interesting parts of this object are `data`, which contains the handwritten digits pixels matrices, and `targets`, which contains the corresponding label for those digits. Both are NumPy arrays.

To get a grasp of what this looks like, we will take the first digit in the data and reshape it into an 8 x 8 matrix; this is the size of the source images. Each value represents a pixel on a grayscale, from 0 to 16.

Then, we print the label of this first digit, which is a 0. If you run this code, you'll get the following output:

```
$ python chapter12/chapter12_load_digits.py
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
0
```

Somehow, we can guess the shape of the zero from the matrix.

Now, let's try to build a model that recognizes handwritten digits. To start simple, we'll use a Gaussian Naive Bayes model, which we'll cover in more detail in the *Classifying data with Naive Bayes models* section. The following example shows the entire process:

chapter12_fit_predict.py

```
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB

digits = load_digits()
```

```
data = digits.data
targets = digits.target

# Split into training and testing sets
training_data, testing_data, training_targets, testing_targets
= train_test_split(
    data, targets, random_state=0
)

# Train the model
model = GaussianNB()
model.fit(training_data, training_targets)

# Run prediction with the testing set
predicted_targets = model.predict(testing_data)

# Compute the accuracy
accuracy = accuracy_score(testing_targets, predicted_targets)
print(accuracy)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_fit_predict.py

Now that we've loaded the dataset, we can see that it takes care of splitting it into a training and a testing set. As we mentioned in the *Model validation* section, this is essential for computing meaningful accuracy scores to check how our model performs.

To do this, we can rely on the `train_test_split` function, which is provided in the `model_selection` package. It selects random instances from our dataset to form the two sets. By default, it keeps 25% percent of the data to create a testing set, but this can be customized. The `random_state` argument allows us to set the random seed to make the example reproducible. You can find out more about this function in the official documentation: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html#sklearn-model-selection-train-test-split.

Then, we must instantiate the `GaussianNB` class. This class is one of the numerous ML estimators that's implemented in scikit-learn. Each has its own set of parameters, to finely tune the behavior of the algorithm. However, scikit-learn is designed to provide sensible defaults for all the estimators, so it's usually good to start with the defaults before tinkering with them.

After that, we must call the `fit` method to train our model. It expects an argument and two arrays: the first one is the actual data, with all its features, while the second one is the corresponding labels. And that's it! You've trained your first ML model!

Now, let's see how it behaves: we'll call `predict` on our model with the testing set so that it automatically classifies the digits of the testing set. The result of this is a new array with the predicted labels.

All we have to do now is compare it with the actual labels of our testing set. Once again, scikit-learn helps by providing the `accuracy_score` function in the `metrics` package. The first argument is the true labels, while the second is the predicted labels.

If you run this code, you'll get an accuracy score of around 83%. That isn't too bad for a first approach! As you have seen, training and running prediction on an ML model is straightforward with scikit-learn.

In practice, we often need to perform pre-processing steps on the data before feeding it to an estimator. Rather than doing this sequentially by hand, scikit-learn proposes a convenient feature that can automate this process: pipelines.

Chaining pre-processors and estimators with pipelines

Quite often, you'll need to pre-process your data so that it can be used by the estimator you wish to use. Typically, you'll want to transform an image into an array of pixel values or, as we'll see in the following example, transform raw text into numerical values so that we can apply some math to them.

Rather than writing those steps by hand, scikit-learn proposes a feature that can automatically chain pre-processors and estimators: **pipelines**. Once created, they expose the very same interfaces as any other estimator, allowing you to run training and prediction in one operation.

To show you what this looks like, we'll look at an example of another classic dataset; that is, the 20 newsgroups text dataset. It consists of 18,000 newsgroup articles categorized into 20 topics. The goal of this dataset is to build a model that will automatically categorize an article in one of those topics.

The following example shows how we can load this data thanks to the `fetch_20newsgroups` function:

chapter12_pipelines.py

```
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

# Load some categories of newsgroups dataset
categories = [
    "soc.religion.christian",
    "talk.religion.misc",
    "comp.sys.mac.hardware",
    "sci.crypt",
]
newsgroups_training = fetch_20newsgroups(
    subset="train", categories=categories, random_state=0
)
newsgroups_testing = fetch_20newsgroups(
    subset="test", categories=categories, random_state=0
)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_pipelines.py

Since the dataset is rather large, we can only load some of the categories. Here, we'll only use four categories. Also, notice that it's already been split into training and testing sets, so we only have to load them with the corresponding argument. You can find out more about the functionality of this dataset in the official documentation: https://scikit-learn.org/stable/datasets/real_world.html#the-20-newsgroups-text-dataset.

Before moving on, it's important to understand what the underlying data is. Actually, this is the raw text of an article. You can check this by printing one of the samples in the data:

```
>>> newsgroups_training.data[0]
"From: sandvik@newton.apple.com (Kent Sandvik)\nSubject:
Re: Ignorance is BLISS, was Is it good that Jesus died?\n
nOrganization: Cookamunga Tourist Bureau\nLines: 17\n\n
nIn article <fl682Ap@quack.kfu.com>, pharvey@quack.kfu.com
(Paul Harvey)\nwrote:\n> In article <sandvik-170493104859@
sandvik-kent.apple.com> \n> sandvik@newton.apple.com (Kent
Sandvik) writes:\n> >Ignorance is not bliss!\n \n> Ignorance
is STRENGTH!\n> Help spread the TRUTH of IGNORANCE!\n\nHuh,
if ignorance is strength, then I won't distribute this piece\n
nof information if I want to follow your advice (contradiction
above) .\n\n\nCheers,\nKent\n---\nsandvik@newton.apple.com.
ALink: KSAND -- Private activities on the net.\n"
```

So, we need to extract some features from this text before feeding it to an estimator. A common approach for this when working with textual data is to use the **Term Frequency-Inverse Document Frequency (TF-IDF)**. Without going into too much detail, this technique will count the occurrences of each word in all the documents (term frequency), weighted by the importance of this word in every document (inverse document frequency). The idea is to give more weight to rarer words, which should convey more sense than frequent words such as "the." You can find out more about this in the scikit-learn documentation: https://scikit-learn.org/dev/modules/feature_extraction.html#tfidf-term-weighting.

This operation consists of splitting each word in the text samples and counting them. Usually, we apply a lot of techniques to refine this, such as removing **stop words**; common words such as "and" or "is" that don't bring much information. Fortunately, scikit-learn provides an all-in-one tool for this: `TfidfVectorizer`.

This pre-processor can take an array of text, tokenize each word, and compute the TF-IDF for each of them. A lot of options are available for finely tuning its behavior, but the defaults are a good start for English text. The following example shows how to use it with an estimator in a pipeline:

chapter12_pipelines.py

```
# Make the pipeline
model = make_pipeline(
    TfidfVectorizer(),
    MultinomialNB(),
)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_pipelines.py

The `make_pipeline` function accepts any number of pre-processors and an estimator in its argument. Here, we're using the Multinomial Naive Bayes classifier, which is suitable for features representing frequency.

Then, we can simply train our model and run prediction to check its accuracy, as we did previously. You can see this in the following example:

chapter12_pipelines.py

```
# Train the model
model.fit(newsgroups_training.data, newsgroups_training.target)

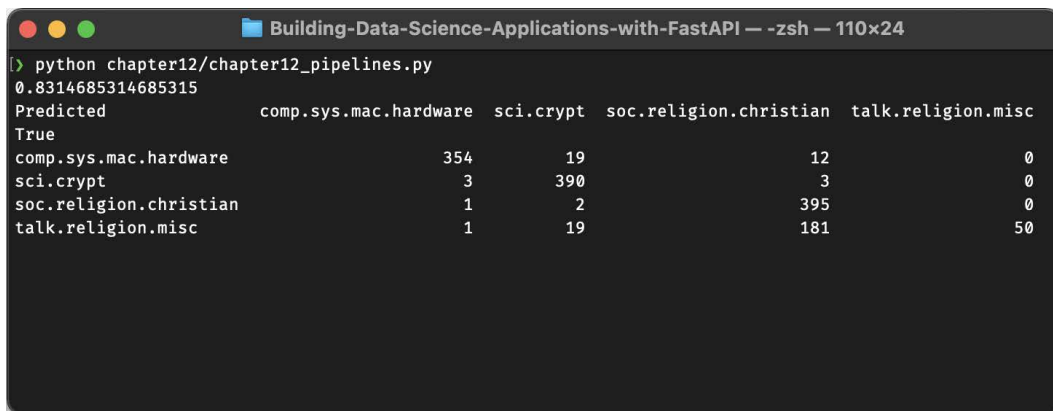
# Run prediction with the testing set
predicted_targets = model.predict(newsgroups_testing.data)

# Compute the accuracy
accuracy = accuracy_score(newsgroups_testing.target, predicted_targets)
print(accuracy)

# Show the confusion matrix
confusion = confusion_matrix(newsgroups_testing.target,
                             predicted_targets)
confusion_df = pd.DataFrame(
    confusion,
    index=pd.Index(newsgroups_testing.target_names,
                   name="True"),
    columns=pd.Index(newsgroups_testing.target_names,
                    name="Predicted"),
)
print(confusion_df)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_pipelines.py

Notice that we also printed a confusion matrix, which is a very convenient representation of the global results. scikit-learn has a dedicated function for this called `confusion_matrix`. Then, we wrap the result in a pandas DataFrame so that we can set the axis labels to improve readability. If you run this example, you'll get an output similar to what's shown in the following screenshot. Depending on your machine and system, it could take a couple of minutes to run:



```

> python chapter12/chapter12_pipelines.py
0.8314685314685315
Predicted      comp.sys.mac.hardware  sci.crypt  soc.religion.christian  talk.religion.misc
True
comp.sys.mac.hardware      354         19         12          0
sci.crypt                   3        390          3          0
soc.religion.christian     1         2        395          0
talk.religion.misc         1         19        181         50

```

Figure 12.3 – Using a confusion matrix on 20 newsgroups dataset

Here, you can see that our results weren't too bad for our first try. Notice that there is one big area of confusion between the *soc.religion.christian* and *talk.religion.misc* categories, which is not very surprising, given their similarity.

As you've seen, building a pipeline with a pre-processor is very straightforward. The nice thing about this is that it automatically applies it to the training data, but also when you're predicting the results.

Before moving on, let's look at one more important feature of scikit-learn: cross-validation.

Validating the model with cross-validation

In the *Model validation* section, we introduced the cross-validation technique, which allows us to use data in training or testing sets. As you may have guessed, this technique is so common that it's implemented natively in scikit-learn!

Let's take another look at the handwritten digit example and apply cross-validation:

chapter12_cross_validation.py

```
from sklearn.datasets import load_digits
from sklearn.model_selection import cross_val_score
from sklearn.naive_bayes import GaussianNB

digits = load_digits()

data = digits.data
targets = digits.target

# Create the model
model = GaussianNB()

# Run cross-validation
score = cross_val_score(model, data, targets)

print(score)
print(score.mean())
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_cross_validation.py

This time, we don't have to split the data ourselves: the `cross_val_score` function performs the folds automatically. In argument, it expects the estimator, `data`, which contains the handwritten digits' pixels matrices, and `targets`, which contains the corresponding label for those digits. By default, it performs five folds.

The result of this operation is an array that provides the accuracy score of the five folds. To get a global overview of this result, we can take, for example, the mean. If you run this example, you'll get the following output:

```
$ python chapter12/chapter12_cross_validation.py
[0.78055556 0.78333333 0.79387187 0.8718663  0.80501393]
0.8069281956050759
```

As you can see, our mean accuracy is around 80%, which is a bit lower than the 83% we obtained with single training and testing sets. That's the main benefit of cross-validation: we obtain a more statistically accurate metric regarding the performance of our model.

With that, we have learned the basics of working with scikit-learn. Before going back to FastAPI, we'll review two categories of ML models: Naive Bayes models and support vector machines.

Classifying data with Naive Bayes models

Even though you probably hear a lot about super-advanced ML methods such as deep learning, it's important to say that simpler methods have existed for years and have proven to be very efficient in many situations. Generally, it's always a good idea when you start with a data science problem to try out simpler models that have fewer parameters and are easier to tune. This will quickly give you a baseline to compare with more advanced techniques.

In this section, we'll review *Naive Bayes models*, a group of fast and simple classification algorithms.

Intuition

Naive Bayes models rely on *Bayes' theorem*, which defines an equation to describe the probability of an event, given the probability of related events. In the context of classification, it gives us an equation to describe the probability of a label, L , given a set of *features*. In our handwritten digit recognition problem, this would translate to "the probability of this observation being the digit *zero*, given the pixel's matrix values." This equation looks like this:

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L) \times P(L)}{P(\text{features})}$$

The notation $P(L \mid \text{features})$ means "the probability of L , given features."

In practice, our classifier will have to decide if an observation has a higher probability of being L_1 or L_2 : "does it look more like a *zero* or an *eight*?" To do this, we can compute the ratio of the two probabilities, which, thanks to the previous equation, gives us the following:

$$\frac{P(L_1 \mid \text{features})}{P(L_2 \mid \text{features})} = \frac{P(\text{features} \mid L_1) \times P(L_1)}{P(\text{features} \mid L_2) \times P(L_2)}$$

The raw probability of L_1 and L_2 , $P(L_1)$ and $P(L_2)$, is the relative frequency of L_1 and L_2 in the training set. If our training set contains 100 samples and we have 15 samples of *zero*, the probability of the label being *zero* is 0.15.

Now, we have to find a way to compute the probability of the features given a label, $P(\text{features} | L_1)$ and $P(\text{features} | L_2)$. What we'll do here is make assumptions about the distribution of the data by finding simple statistical rules. This is why those models are called "naïve."

One of the first classical assumptions regarding those models is Gaussian distribution.

Classifying data with Gaussian Naive Bayes

As we mentioned previously, Naive Bayes models work by making "naive" assumptions about the distribution of the underlying data. In the case of Gaussian Naive Bayes, we assume that the data is drawn from a *Gaussian distribution* (or normal distribution). The following is a graphical representation of such a distribution:

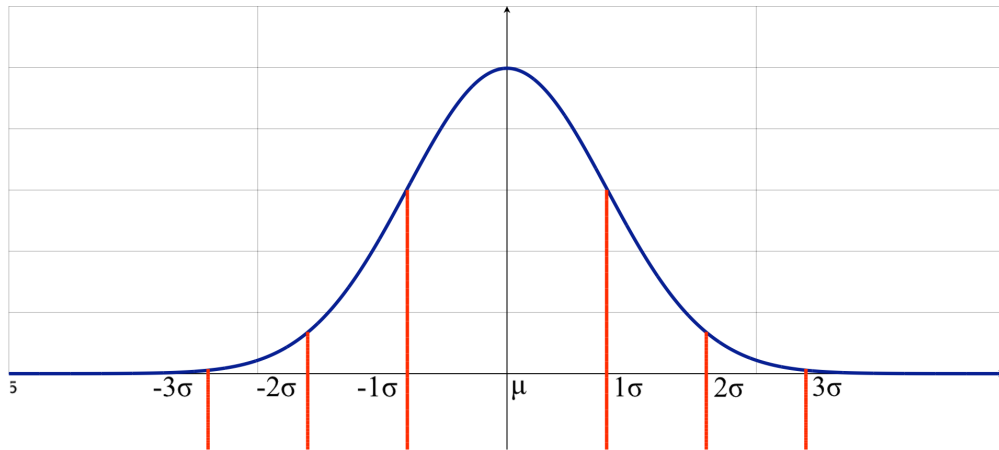


Figure 12.4 – Curve of a Gaussian distribution

The intuition behind this is that, for data following a Gaussian distribution, the probability is high around the mean, μ , and the standard deviation, σ . It then decreases rapidly when it moves away from the mean. This is computed using the following formula:

$$\frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

Then, all we need to do to train our model is *compute the mean and standard deviation for each feature in each label*. This will give us, for each label, a simple formula to compute the probability of having features, given L . Once we have them, all we need to do is apply the preceding formula to get the probability of this observation, given L .

This is exactly what happens when we train the GaussianNB estimator in scikit-learn. If we consider the same example we showed in the *Training models and predicting* section, we can retrieve the mean and standard deviation that was computed for each pixel for each possible digit. In the following example, you can see that we are training a Gaussian Naive Bayes model with the handwritten digits set, before printing the mean and standard deviation for the digit zero:

chapter12_gaussian_naive_bayes.py

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB

digits = load_digits()

data = digits.data
targets = digits.target

# Split into training and testing sets
training_data, testing_data, training_targets, testing_targets
= train_test_split(
    data, targets, random_state=0
)

# Train the model
model = GaussianNB()
model.fit(training_data, training_targets)

# Print mean and standard deviation of digit zero
print("Mean of each pixel for digit zero")
print(model.theta_[0])
```

```
print("Standard deviation of each pixel for digit zero")
print(model.sigma_[0])
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_gaussian_naive_bayes.py

If you run this example, you'll get the following output:

```
$ python chapter12/chapter12_gaussian_naive_bayes.py
Mean of each pixel for digit zero
[0.00000000e+00 2.83687943e-02 4.12765957e+00 1.29716312e+01
 1.13049645e+01 2.96453901e+00 3.54609929e-02 0.00000000e+00
 0.00000000e+00 9.50354610e-01 1.25035461e+01 1.37021277e+01
 1.16453901e+01 1.12765957e+01 9.00709220e-01 0.00000000e+00
 0.00000000e+00 3.79432624e+00 1.43758865e+01 5.57446809e+00
 2.13475177e+00 1.23049645e+01 3.43971631e+00 0.00000000e+00
 0.00000000e+00 5.31205674e+00 1.27517730e+01 2.06382979e+00
 1.34751773e-01 9.26241135e+00 6.45390071e+00 0.00000000e+00
 0.00000000e+00 5.78723404e+00 1.16737589e+01 1.00000000e+00
 5.67375887e-02 8.89361702e+00 7.10638298e+00 0.00000000e+00
 0.00000000e+00 3.41843972e+00 1.33687943e+01 1.82269504e+00
 1.69503546e+00 1.12127660e+01 5.90070922e+00 0.00000000e+00
 0.00000000e+00 7.80141844e-01 1.29787234e+01 1.02056738e+01
 1.06382979e+01 1.32340426e+01 2.53191489e+00 0.00000000e+00
 0.00000000e+00 7.09219858e-03 4.15602837e+00 1.35602837e+01
 1.33049645e+01 5.46099291e+00 2.83687943e-01 0.00000000e+00]
Standard deviation of each pixel for digit zero
[4.30146180e-08 5.59328432e-02 9.13263925e+00 5.40345057e+00
 1.19566421e+01 1.10838489e+01 3.42035539e-02 4.30146180e-08
 4.30146180e-08 3.62164885e+00 1.24060158e+01 8.98928630e+00
 1.66827625e+01 1.22284594e+01 3.08233997e+00 4.30146180e-08
 4.30146180e-08 7.09954232e+00 5.32679447e+00 2.42870077e+01
 1.03435441e+01 1.03112520e+01 7.16835174e+00 4.30146180e-08
 4.30146180e-08 6.08701780e+00 1.01298728e+01 1.13505357e+01
 3.57728527e-01 1.27609276e+01 5.38262667e+00 4.30146180e-08
 4.30146180e-08 5.03274487e+00 1.11843469e+01 5.54609933e+00]
```


1.38624861e-01	1.46624416e+01	7.28655505e+00	4.30146180e-08
4.30146180e-08	5.17951818e+00	5.96328157e+00	9.69196725e+00
8.97791866e+00	1.45362910e+01	1.38482974e+01	4.30146180e-08
4.30146180e-08	1.80272626e+00	7.62366082e+00	1.54257835e+01
1.74365475e+01	1.00516071e+01	1.00503999e+01	4.30146180e-08
4.30146180e-08	7.04194232e-03	7.77707363e+00	4.30310351e+00
7.87153568e+00	1.51846487e+01	9.83350981e-01	4.30146180e-08]

All those numbers represent the means and standard deviations of the 64 pixels of the 8x8 pixel matrix, for the digit zero.

If you want to learn more about the mathematics behind this, you can read a very detailed introduction in the following PennState online course: <https://online.stat.psu.edu/stat414/lesson/16>.

This is why training and running prediction on a Gaussian Naive Bayes model is so fast: it only involves simple mathematical computations. Of course, its accuracy only depends on the correctness of the assumption: if our data doesn't conform to a Gaussian distribution, the model won't perform very well. Still, its simplicity and efficiency always make it a good basis before we consider more complex algorithms.

Classifying data with Multinomial Naive Bayes

Another assumption we can make about the data is that it follows a *multinomial distribution*. This is particularly suited for datasets with *features representing counts*, such as the number of times they appear in the dataset, such as word frequencies.

If we consider some text and we compute the frequency of each word (or the TF-IDF, as we saw in the *Chaining pre-processors and estimators with pipelines* section), how do we compute its probability of being in the L category; that is, our famous $P(\text{features} | L)$? The multinomial law says that it can be computed using this formula:

$$\frac{n!}{x_1! x_2! \dots x_i!} p_1^{x_1} p_2^{x_2} \dots p_i^{x_i}$$

Here, n is the total number of occurrences, x_1, x_2, \dots, x_i is the number of occurrences of the word 1, 2... i , and p_1, p_2, \dots, p_i is the probability of the word 1, 2... i .

All we need to do now is find the probability of each word in each category: this is the purpose of the training phase. It's computed as follows:

$$p_{Li} = \frac{N_{Li} + \alpha}{N_L + \alpha n}$$

Here, N_{Li} is the frequency of the word i in category L , N_L is the total number of occurrences of every word in category L , and n the number of different words. α is a smoothing parameter to prevent some probabilities from being equal to zero, which would then propagate in the multinomial formula. It's usually set to 1 by default, but this can be tuned.

If you want to learn more about the mathematics behind this, you can read a very detailed introduction to it in the following PennState online course: <https://online.stat.psu.edu/stat504/lesson/1/1.7>.

When training a `MultinomialNB` estimator with scikit-learn, this is exactly what the algorithm does: it computes the probability of each word in each category.

When predicting the category of a new piece of text, it simply has to count the frequency of each word and apply the first formula with the probabilities it computed during training.

That's it for the theory behind Naive Bayes models. The key thing to remember is that they are *very fast to train* and generally provide quite a *good basis when starting with a classification problem*. Besides, they tend to work quite well if the number of features is large.

In the next section, we'll review another type of model that's quite powerful both for classification and regression: support vector machines.

Classifying data with support vector machines

Support Vector Machines (SVM) are another group of classification and regression models that have proven to be quite powerful in many situations. The intuition behind them is quite straightforward to understand, but we'll see that their power comes mostly from a mathematical technique that's used in many other ML algorithms, called the **kernel trick**.

Intuition

Let's consider a simple classification problem where we want to classify samples into two categories. The following is a graph containing some randomly generated data for this problem:

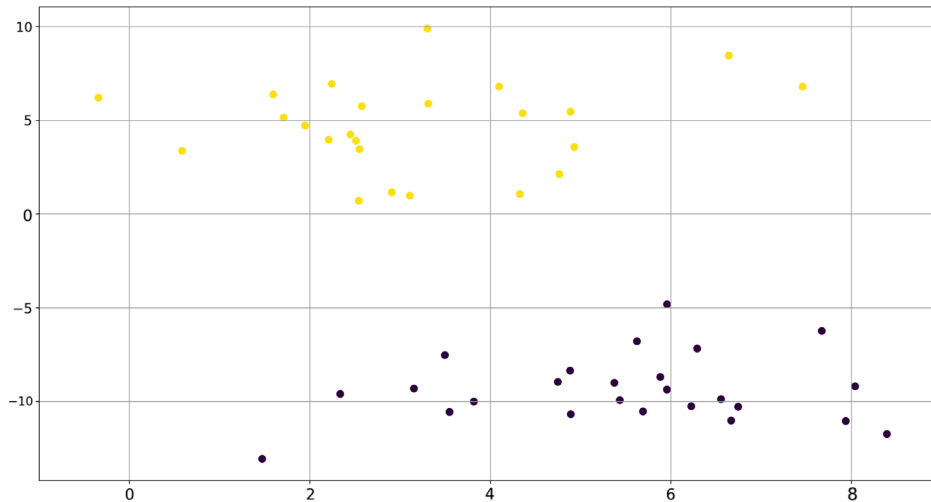


Figure 12.5 – Simple classification problem data

Intuitively, with such data, finding a straight line to cleanly separate the two categories seems simple. However, we quickly see that there are a lot of different solutions, as shown in the following graph:

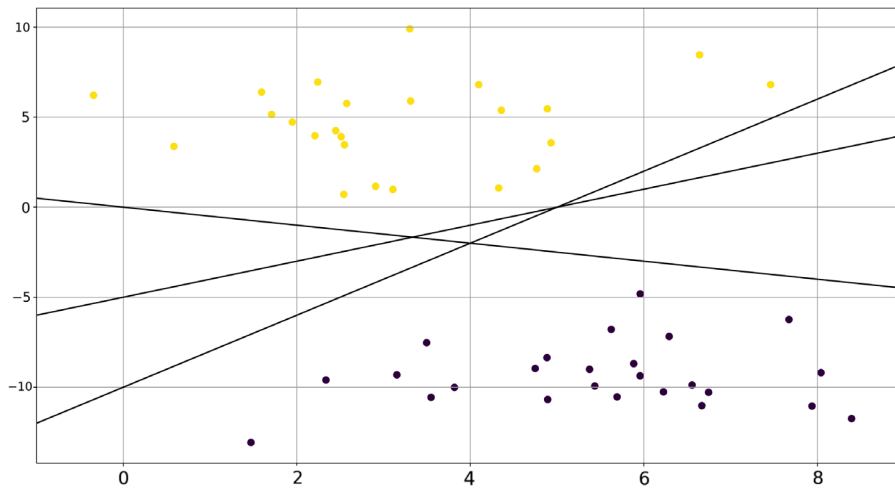


Figure 12.6 – Three possible linear classifiers

So, how do we find the one that will yield the best results to predict the category of a new point?

What SVM does is draw a *margin* around each of those possible classifiers, up to the nearest point. The classifier that maximizes the margin is the one that'll be selected for our model. If we train an SVM on our sample dataset, we'll obtain the classifier shown in the following graph. This graph also shows the margin for better visualization:

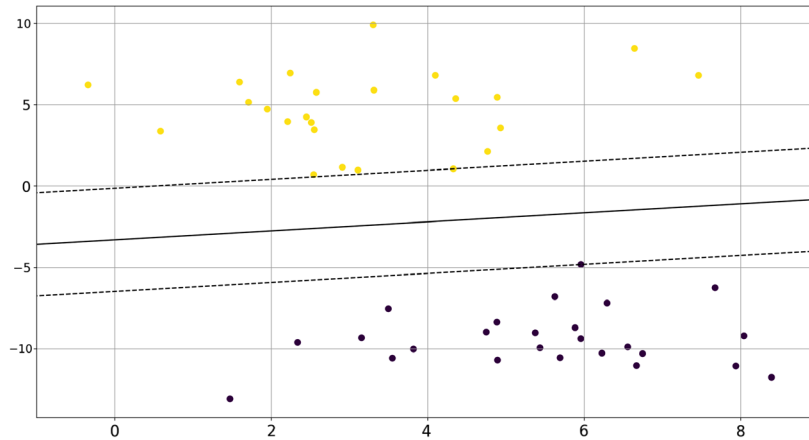


Figure 12.7 – Three possible linear classifiers

The two samples that are touching the margin are the *support vectors*.

Of course, in the real world, having such nicely separated data is very rare, and a linear classifier may not exist. The following graph shows some randomly generated data that is not linearly separable:

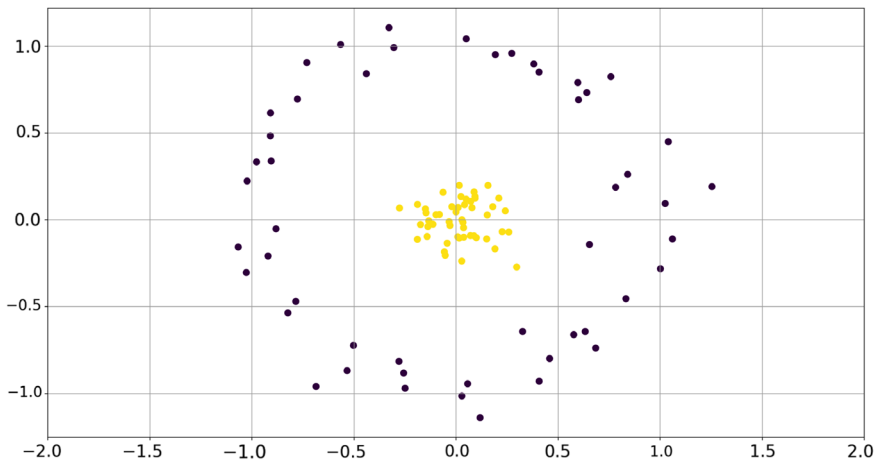


Figure 12.8 – Non-linearly separable data

To solve this, SVM projects the dataset to a higher dimension by applying a kernel function to the data. We won't go into the mathematical details of this, but kernel functions can compute the similarity between each pair of points: in the new dimension, similar points are close, while dissimilar points are distant.

Metaphorically, imagine that we draw the data shown in the preceding graph on a sheet of paper. The goal of the kernel is to find a way to fold or bend this paper so that the yellow and purple dots can be linearly separated by a plane.

Several kernel functions exist, such as the **Radial Basis Function (RBF)**, which is applied by default when using SVM with scikit-learn.

The following graph shows the result of performing such an operation on our sample data:

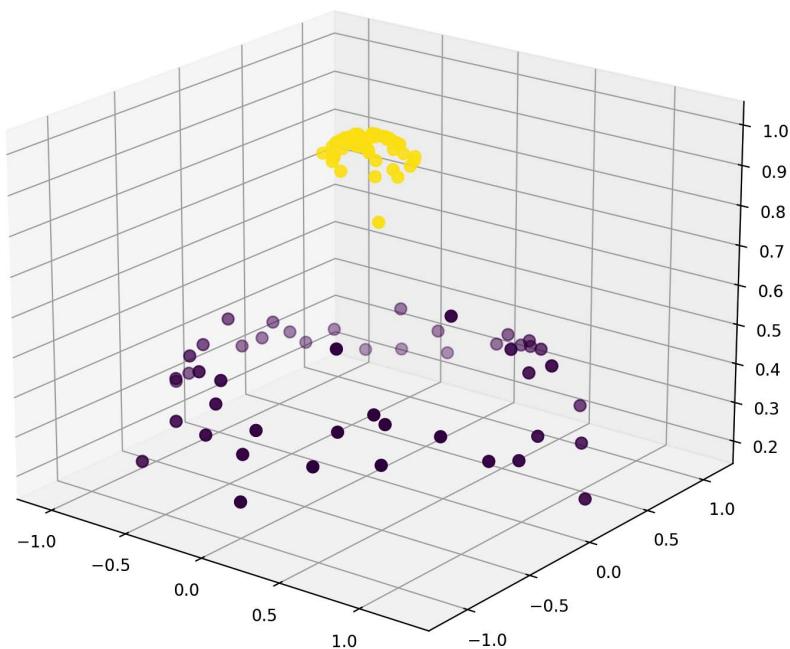


Figure 12.9 – Data projected in a third dimension that's now linearly separable

Here, we can see that there is a clear linear classifier in three dimensions that can separate the data.

You can read more about the mathematics behind this in the following scikit-learn documentation: <https://scikit-learn.org/stable/modules/svm.html#mathematical-formulation>.

Using SVM in scikit-learn

Now that we have a good grasp of the functionality of SVM, we can try using it in scikit-learn. As you'll see, it's not very different from what we've seen so far with Naive Bayes models.

It comes in different flavors, with slight adaptations depending on your use case. Typically, the *SVC* estimator is suitable for classification problems, while *SVR* is usually adapted to regression.

In the following example, once again, we're taking our handwritten digit recognition example and applying the *SVC* estimator. We will evaluate it using the cross-validation method:

chapter12_svm.py

```
from sklearn.datasets import load_digits
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

digits = load_digits()

data = digits.data
targets = digits.target

# Create the model
model = SVC()

# Run cross-validation
score = cross_val_score(model, data, targets)

print(score)
print(score.mean())
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_svm.py

As you can see, we simply instantiate the `SVC` class and keep the default parameters. If you run this example, you'll get the following output:

```
$ python chapter12/chapter12_svm.py
[0.96111111 0.94444444 0.98328691 0.98885794 0.93871866]
0.9632838130609718
```

The mean accuracy of our model is 96%! That's quite impressive, given that we didn't even have to tune the parameters.

Finding the best parameters

With Naive Bayes models, we almost had no parameters to tune. In the case of SVM, however, there are quite a few of them – most notably, there's the kernel function, which is RBF by default, and the C parameter. C defines the "hardness" of the margin around the linear classifier: if C is high, no point can creep inside the margin. A lower C will relax this constraint and, in some cases, allow a better fit for the data.

However, finding the best set of parameters is not always intuitive and it would be quite time-consuming to do so by hand. What can we do, then? scikit-learn can help us with this!

The `model_selection` package provides a useful class called `GridSearchCV` that allows us to automatically search for the best parameters for our estimator. Here, we set the different parameters we want to try and it trains the model with every possible combination. At the end of this process, it returns the parameters that achieved the best accuracy.

In the following example, we implemented a grid search to find the best parameters for C and the kernel function for our handwritten digit recognition problem:

chapter12_finding_parameters.py

```
from sklearn.datasets import load_digits
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

digits = load_digits()

data = digits.data
targets = digits.target
```

```
# Create the grid of parameters
param_grid = {
    "C": [1, 10, 100, 1000],
    "kernel": ["linear", "poly", "rbf", "sigmoid"]
}
grid = GridSearchCV(SVC(), param_grid)

grid.fit(data, targets)

print("Best params", grid.best_params_)
print("Best score", grid.best_score_)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter12/chapter12_finding_parameters.py

As you can see, we only have to create a dictionary for mapping the name of the parameter to the list of values we want to try for this parameter. `GridSearchCV` is then initialized with the estimator instance and this parameter grid.

Calling the `fit` method with the dataset will run the search. Once done, you'll have access to the `best_params_` and `best_score_` properties, which will give you the best results.

If you run this example, you'll get the following result:

```
$ python chapter12/chapter12_finding_parameters.py
Best params {'C': 10, 'kernel': 'rbf'}
Best score 0.9738502011761063
```

Here, we achieved 97% accuracy with the `C` parameter set to 10 and the RBF kernel function.

Of course, the larger your grid is, the more time you'll need to compute all the possibilities. If you have a very large set of parameters to try, have a look at `RandomizedSearchCV`, which works similarly but only tests a few combinations by picking some randomly. You can learn more about this in the scikit-learn documentation: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html.

Summary

Congratulations! You've discovered the basics concepts of ML and scikit-learn. Now, you should be able to explore your first data science problems in Python. Of course, this was by no means a complete lesson on ML: the field is vast and there are tons of algorithms and techniques to explore. However, I hope that this has sparked your curiosity and that you'll deepen your knowledge of this subject.

Now, it's time to get back to FastAPI! With our new ML tools at hand, we'll be able to leverage the power of FastAPI to serve our estimators and propose a reliable and efficient prediction API for our users.

13

Creating an Efficient Prediction API Endpoint with FastAPI

In the previous chapters, we introduced the most common data science techniques and libraries largely used in the Python community. Thanks to those tools, we can now build machine learning models that can make efficient predictions and classify data. Of course, we now have to think about a convenient interface so that we can take advantage of their intelligence. This way, microservices or frontend applications can ask our model to make predictions to improve the user experience or business operations.

In this chapter, we'll learn how to do that with FastAPI. As we've seen throughout this book, FastAPI allows us to implement very efficient REST APIs with clear and lightweight syntax. In this chapter, you'll learn how to do this as efficiently as possible so that it can serve thousands of prediction requests. To help us with this task, we'll introduce another library, Joblib, that provides tools to help us serialize a trained model and cache predicted results.

In this chapter, we're going to cover the following main topics:

- Persisting a trained model with Joblib
- Implementing an efficient prediction endpoint
- Caching results with Joblib

Technical requirements

You'll need a Python virtual environment, similar to the one we set up in *Chapter 1, Python Development Environment Setup*.

You can find all the code examples for this chapter in this book's dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter13>.

Persisting a trained model with Joblib

In the previous chapter, you learned how to train an estimator with scikit-learn. When building such models, you'll likely obtain a rather complex Python script to load your training data, pre-process it, and train your model with the best set of parameters. However, when deploying your model in a web application, such as FastAPI, you don't want to repeat this script and run all those operations when the server is starting. Instead, you need a ready-to-use representation of your trained model that you can just load and use.

This is what Joblib does. This library aims to provide tools for efficiently saving Python objects to disk, such as large arrays of data or function results: this operation is generally called **dumping**. Joblib is already a dependency of scikit-learn, so we don't even need to install it. scikit-learn uses it internally to load the bundled toy datasets.

As we'll see, dumping a trained model involves just one line of code with Joblib.

Dumping a trained model

In this example, we're using the newsgroups example we saw in the *Chaining pre-processors and estimators with pipelines* section of *Chapter 12, Training Machine Learning Models with scikit-learn*. As a reminder, we load four categories of the 20 newsgroups dataset and build a model to automatically categorize news articles into those categories. Once we've done this, we dump the model into a file called `newsgroups_model.joblib`:

chapter13_dump_joblib.py

```
# Make the pipeline
model = make_pipeline(
    TfidfVectorizer(),
    MultinomialNB(),
)

# Train the model
model.fit(newsgroups_training.data, newsgroups_training.target)

# Serialize the model and the target names
model_file = "newsgroups_model.joblib"
model_targets_tuple = (model, newsgroups_training.target_names)
joblib.dump(model_targets_tuple, model_file)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_dump_joblib.py

As you can see, Joblib exposes a function called `dump`, which simply expects two arguments: the Python object to save and the path of the file.

Notice that we don't dump the `model` variable alone: instead, we wrap it in a tuple, along with the name of the categories, `target_names`. This allows us to retrieve the actual name of the category after the prediction has been made, without us having to reload the training dataset.

If you run this script, you'll see that the `newsgroups_model.joblib` file was created:

```
$ python chapter13/chapter13_dump_joblib.py
$ ls -lh *.joblib
-rw-r--r-- 1 fvoron staff 3,2M 10 jul 10:41 newsgroups_
model.joblib
```

Notice that this file is rather large: it's more than 3 MB! It stores all the probabilities of each word in each category, as computed by the Multinomial Naive Bayes model.

That's all we need to do. This file now contains a static representation of our Python model, which will be easy to store, share, and load. Now, let's learn how to load it and check that we can run predictions on it.

Loading a dumped model

Now that we have our dumped model file, let's learn how to load it again using Joblib and check that everything is working. In the following example, we're loading the Joblib dump present in the `chapter13` directory of the `examples` repository and running a prediction:

`chapter13_load_joblib.py`

```
import os
from typing import List, Tuple

import joblib
from sklearn.pipeline import Pipeline

# Load the model
model_file = os.path.join(os.path.dirname(__file__),
                           "newsgroups_model.joblib")
loaded_model: Tuple[Pipeline, List[str]] = joblib.load(model_file)
model, targets = loaded_model

# Run a prediction
p = model.predict(["computer cpu memory ram"])
print(targets[p[0]])
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_load_joblib.py

All we need to do here is call the `load` function from Joblib and pass it a valid path to a dump file. The result of this function is the very same Python object we dumped. Here, it's a tuple composed of the scikit-learn estimator and a list of categories.

Notice that we added some type hints: while not necessary, it helps mypy or your IDE identify the nature of the objects you loaded and benefit from type-checking and auto-completion.

Finally, we ran a prediction on the model: it's a true scikit-learn estimator, with all the necessary training parameters.

That's it! As you've seen, Joblib is straightforward to use. Nevertheless, it's an essential tool for exporting your scikit-learn models and being able to use them in external services without repeating the training phase. Now, we can use those dump files in FastAPI projects.

Implementing an efficient prediction endpoint

Now that we have a way to save and load our machine learning models, it's time to use them in a FastAPI project. As you'll see, the implementation shouldn't be too much of a surprise if you've followed this book. The main part of the implementation is the class dependency, which will take care of loading the model and making predictions. If you need a refresher on class dependencies, check out *Chapter 5, Dependency Injections in FastAPI*.

Let's go! Our example will be based on the newsgroups model we dumped in the previous section. We'll start by showing you how to implement the class dependency, which will take care of loading and making predictions:

chapter13_prediction_endpoint.py

```
class PredictionInput(BaseModel):
    text: str

class PredictionOutput(BaseModel):
    category: str

class NewsgroupsModel:
    model: Optional[Pipeline]
    targets: Optional[List[str]]

    def load_model(self):
        """Loads the model"""
        model_file = os.path.join(os.path.dirname(__file__),
            "newsgroups_model.joblib")
        loaded_model: Tuple[Pipeline, List[str]] = joblib.
```

```
load(model_file)
    model, targets = loaded_model
    self.model = model
    self.targets = targets

    async def predict(self, input: PredictionInput) ->
    PredictionOutput:
        """Runs a prediction"""
        if not self.model or not self.targets:
            raise RuntimeError("Model is not loaded")
        prediction = self.model.predict([input.text])
        category = self.targets[prediction[0]]
        return PredictionOutput(category=category)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_prediction_endpoint.py

First, we start by defining two Pydantic models: `PredictionInput` and `PredictionOutput`. In a pure FastAPI philosophy, they will help us validate the request payload and return a structured JSON response. Here, as input, we simply expect a `text` property containing the text we want to classify; in terms of the output, we expect a `category` property containing the predicted category.

The most interesting part of this extract is the `NewsGroupsModel` class. It implements two methods: `load_model` and `predict`.

The `load_model` method loads the model using `Joblib`, as we saw in the previous section, and stores the model and the targets in class properties. Hence, they will be available for use by the `predict` method.

On the other hand, the `predict` method will be injected into the path operation function. As you can see, it directly accepts a `PredictionInput` that will be injected by FastAPI. Inside this method, we are making a prediction, as we usually do with `scikit-learn`. We return a `PredictionOutput` object with the category we predicted.

You may have noticed that, first, we check if the model and its targets have been assigned in the class properties before performing the prediction. Of course, we need to ensure `load_model` was called at some point before making a prediction. You may be wondering why we are not putting this logic in an initializer, `__init__`, so that we can ensure the model is loaded at class instantiation. This would work perfectly fine; however, it would cause some issues. As we'll see, we are instantiating a `NewsgroupsModel` instance right after FastAPI so that we can use it in our routes. If the loading logic was in `__init__`, the model would be loaded whenever we import some variables (such as the `app` instance) from this file, such as in unit tests. In most cases, this would incur unnecessary I/O operations and memory consumption. As we'll see, it's better to use the startup event of FastAPI to load the model when the app is run.

The following extract shows the rest of the implementation, along with the actual FastAPI route for handling predictions:

chapter13_prediction_endpoint.py

```
app = FastAPI()
newsgroups_model = NewsgroupsModel()

@app.post("/prediction")
async def prediction(
    output: PredictionOutput = Depends(newsgroups_model.
predict),
) -> PredictionOutput:
    return output

@app.on_event("startup")
async def startup():
    newsgroups_model.load_model()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_prediction_endpoint.py

As we mentioned previously, we are creating an instance of `NewsgroupsModel` so that we can inject it into our path operation function. Moreover, we are implementing a startup event handler to call `load_model`. This way, we are making sure that the model is loaded during application startup and is ready to use.

The prediction endpoint is quite straightforward: as you can see, we directly depend on the `predict` method, which will take care of injecting the payload and validating it. We only have to return the output.

That's it! Once again, FastAPI makes our life very easy by allowing us to write very simple and readable code, even for complex tasks. We can run this application using Uvicorn, as usual:

```
$ uvicorn chapter13.chapter13_prediction_endpoint:app
```

Now, we can try to run some predictions with HTTPie:

```
$ http POST http://localhost:8000/prediction text="computer cpu
memory ram"
HTTP/1.1 200 OK
content-length: 36
content-type: application/json
date: Tue, 13 Jul 2021 06:34:58 GMT
server: uvicorn
{
  "category": "comp.sys.mac.hardware"
}
```

Our machine learning classifier is alive! To push this further, let's see how we can implement a simple caching mechanism using Joblib.

Caching results with Joblib

If your model takes time to make predictions, it may be interesting to cache the results: if the prediction for a particular input has already been done, it makes sense to return the same result we saved on disk, rather than running the computations again. In this section, we'll learn how to do this with the help of Joblib.

Joblib provides us with a very convenient and easy-to-use tool to do this, so the implementation is quite straightforward. The main concern will be about whether we should choose standard or async functions to implement the endpoints and dependencies. This will allow us to explain some of the technical details of FastAPI in more detail.

We'll build upon the example we provided in the previous section. The first thing we must do is initialize a Joblib `Memory` class, which is the helper for caching functions results. Then, we can add a decorator to the functions we want to cache. You can see this in the following example:

chapter13_caching.py

```
memory = joblib.Memory(location="cache.joblib")

@memory.cache(ignore=["model"])
def predict(model: Pipeline, text: str) -> int:
    prediction = model.predict([text])
    return prediction[0]
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_caching.py

When initializing `memory`, the main argument is `location`, which is the directory path where Joblib will store the results. Joblib automatically saves cached results on the hard disk.

Then, you can see that we implemented a `predict` function that accepts our scikit-learn model, some text input, and then returns the predicted category index. This is the same prediction operation we've seen so far. Here, we extracted it from the `NewsgroupsModel` dependency class because Joblib caching is primarily designed to work with regular functions. Caching class methods is not recommended. As you can see, we simply have to add a `@memory.cache` decorator on top of this function to enable Joblib caching.

Whenever this function is called, Joblib will check if it has the result on disk for the same arguments. If it does, it returns it directly. Otherwise, it proceeds with the regular function call.

As you can see, we added an `ignore` argument to the decorator, which allows us to tell Joblib to not take into account some arguments in the caching mechanism. Here, we excluded the `model` argument. Joblib cannot dump complex objects, such as scikit-learn estimators. This isn't a problem, though: the model is not changing between several predictions, so we don't care about having it cached. If we make improvements to our model and deploy a new one, all we have to do is clear the whole cache so that older predictions are made again with the new model.

Now, we can tweak the `NewsgroupsModel` dependency class so that it works with this new `predict` function. You can see this in the following example:

chapter13_caching.py

```
class NewsgroupsModel:
    model: Optional[Pipeline]
    targets: Optional[List[str]]

    def load_model(self):
        """Loads the model"""
        model_file = os.path.join(os.path.dirname(__file__),
            "newsgroups_model.joblib")
        loaded_model: Tuple[Pipeline, List[str]] = joblib.
load(model_file)
        model, targets = loaded_model
        self.model = model
        self.targets = targets

    def predict(self, input: PredictionInput) ->
PredictionOutput:
        """Runs a prediction"""
        if not self.model or not self.targets:
            raise RuntimeError("Model is not loaded")
        prediction = predict(self.model, input.text)
        category = self.targets[prediction]
        return PredictionOutput(category=category)
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_caching.py

In the `predict` method, we are calling the external `predict` function instead of doing so directly inside the method, taking care to pass the model and the input text as arguments. All we have to do after is retrieve the corresponding category name and build a `PredictionOutput` object.

Finally, we have the REST API endpoints. Here, we added a `DELETE/cache` route so that we can clear the whole Joblib cache with an HTTP request. This can be seen in the following example:

chapter13_caching.py

```
@app.post("/prediction")
def prediction(
    output: PredictionOutput = Depends(newgroups_model.
predict),
) -> PredictionOutput:
    return output

@app.delete("/cache", status_code=status.HTTP_204_NO_CONTENT)
def delete_cache():
    memory.clear()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_caching.py

The `clear` method of the `memory` object removes all the Joblib cache files on the disk.

Our FastAPI application is now caching prediction results. If you make a request with the same input twice, the second response will show you the cached result. In this example, our model is fast, so you won't notice a difference in terms of execution time; however, this could be interesting with more complex models.

Choosing between standard or async functions

You may have noticed that we changed the `predict` method and the `prediction` and `delete_cache` path operation functions so that they're *standard, non-async* functions.

Since the beginning of this book, we've shown you how FastAPI completely embraces asynchronous I/O and why it's good for the performance of your applications. We've also recommended libraries that also work asynchronously, such as database drivers, to leverage that power.

In some cases, however, that's not always possible. In this case, Jobjlib is implemented to work synchronously. Nevertheless, it's performing long I/O operations: it reads and writes cache files on the hard disk. Hence, it will block the process and won't be able to answer other requests while this is happening, as we've explained in the *Asynchronous I/O* section of *Chapter 2, Python Programming Specificities*.

To solve this, FastAPI implements a neat mechanism: *if you define a path operation function or a dependency as a standard, non-async function, it'll run it in a separate thread*. This means that blocking operations, such as synchronous file reading, won't block the main process. In a sense, we could say that it mimics an asynchronous operation.

To understand this, we'll perform a simple experiment. In the following example, we are building a dummy FastAPI application with three endpoints:

- `/fast`, which directly returns a response.
- `/slow-async`, a path operation defined as `async`, which makes a synchronous blocking operation that takes 10 seconds to run.
- `/slow-sync`, a path operation that's defined as a standard method, which makes a synchronous blocking operation that takes 10 seconds to run:

chapter13_async_not_async.py

```
import time

from fastapi import FastAPI

app = FastAPI()

@app.get("/fast")
async def fast():
    return {"endpoint": "fast"}

@app.get("/slow-async")
async def slow_async():
    """Runs in the main process"""
    time.sleep(10) # Blocking sync operation
    return {"endpoint": "slow-async"}
```

```
@app.get("/slow-sync")
def slow_sync():
    """Runs in a thread"""
    time.sleep(10) # Blocking sync operation
    return {"endpoint": "slow-sync"}
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter13/chapter13_async_not_async.py

With this simple application, the goal is to see how those blocking operations block the main process. Let's run this application with Uvicorn:

```
$ uvicorn chapter13.chapter13_async_not_async:app
```

Next, open two new terminals. In the first one, make a request to the /slow-async endpoint:

```
$ http GET http://localhost:8000/slow-async
```

Without waiting for the response, in the second terminal, make a request to the /fast endpoint:

```
$ http GET http://localhost:8000/fast
```

You'll see that you have to wait 10 seconds before you get the response for the /fast endpoint. This means that /slow-async blocked the process and prevented the server from answering the other request while this was happening.

Now, let's perform the same experiment with the /slow-sync endpoint:

```
$ http GET http://localhost:8000/slow-sync
```

And again, run the following command:

```
$ http GET http://localhost:8000/fast
```

You'll immediately get /fast as a response, without having to wait for /slow-sync to finish. Since it's defined as a standard, non-async function, FastAPI will run it in a thread to prevent blocking. However, bear in mind that sending the task to a separate thread implies a small overhead, so it's important to think about the best approach for your current problem.

So, when developing with FastAPI, how can you choose between standard or async functions for path operations and dependencies? The rules of thumb for this are as follows:

- If it's not making long I/O operations (file reading, network requests, and so on), define them as `async`.
- If you are making I/O operations, do the following:
 - a. Try to choose libraries that are compatible with asynchronous I/O, as we saw for databases or HTTP clients. In this case, your functions will be `async`.
 - b. If it's not possible, which is the case for Joblib caching, define them as standard functions. FastAPI will run them in a separate thread.

Since Joblib is completely synchronous at making I/O operations, we switched the path operations and the dependency method so that they're synchronous, standard methods.

In this example, the difference is not very noticeable because the I/O operations are small and fast. However, it's good to keep this in mind if you have to implement slower operations, such as for performing file uploads to cloud storage.

Summary

Congratulations! You're now able to build a fast and efficient REST API to serve your machine learning models. Thanks to Joblib, you've learned how to dump a trained scikit-learn estimator into a file that's easy to load and use inside your application. We've also seen an approach to caching prediction results using Joblib. Finally, we discussed how FastAPI handles synchronous operations by sending them to a separate thread to prevent blocking. While this was a bit technical, it's important to bear this aspect in mind when dealing with blocking I/O operations.

We're near the end of our FastAPI journey. Before letting you build awesome data science applications by yourself, we have provided one last chapter to push this a bit further: using WebSockets and a library dedicated to computer vision, OpenCV, we'll learn how to implement an application that can perform real-time face detection.

14

Implement a Real-Time Face Detection System Using WebSockets with FastAPI and OpenCV

In the previous chapter, you learned how to create efficient REST API endpoints to make predictions with trained machine learning models. This approach covers a lot of use cases, given that we have a single observation we want to work on. In some cases, however, we may need to continuously perform predictions on a stream of input, for instance, a face detection system that works in real time with video input. This is exactly what we'll build in this chapter. How? If you remember, besides HTTP endpoints, FastAPI also has the ability to handle WebSockets endpoints, which allow us to send and receive streams of data. In this case, the browser will send into the WebSocket a stream of images from the webcam, and our application will run a face detection algorithm and send back the coordinates of the detected face in the image. For this face detection task, we'll rely on OpenCV, which is a library dedicated to computer vision.

In this chapter, we're going to cover the following main topics:

- Getting started with OpenCV
- Implementing an HTTP endpoint to perform face detection on a single image
- Implementing a WebSocket to perform face detection on a stream of images
- Sending a stream of images from the browser in a WebSocket
- Showing the face detection results in a browser

Technical requirements

You'll need a Python virtual environment, as we set up in *Chapter 1, Python Development Environment Setup*.

You'll also need a webcam on your computer to be able to run the examples.

You'll find all the code examples of this chapter in the dedicated GitHub repository: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/tree/main/chapter14>.

Getting started with OpenCV

Computer vision is a field related to machine learning that aims at developing algorithms and systems to analyze images and videos automatically. A typical example of computer vision's application is face detection: a system automatically detecting human faces in an image. This is the kind of system we'll build in this chapter.

To help us in this task, we'll use OpenCV, which is one of the most popular computer vision libraries. It's written in C and C++ but has bindings to make it usable in many other programming languages, including Python. We could have used scikit-learn to develop a face detection model, but we'll see that OpenCV already includes all the necessary tools to perform this task without having to manually train and tune machine learning estimators.

To begin with OpenCV, we'll implement a simple Python script to perform face detection locally using a computer webcam:

1. The first step is, of course, to install the OpenCV library for Python:

```
$ pip install opencv-python
```

Now, all we need to do is to use the tools provided by OpenCV to implement a simple face detection program. As you'll see, everything is bundled in the library.

2. In the following example, you can see the whole implementation:

chapter14_opencv.py

```
import cv2

# Load the trained model
face_cascade = cv2.CascadeClassifier(
    cv2.data.harcascades + "haarcascade_frontalface_default.
xml"
)

# You may need to change the index depending on your computer
and camera
video_capture = cv2.VideoCapture(0)

while True:
    # Get an image frame
    ret, frame = video_capture.read()

    # Convert it to grayscale and run detection
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)

    # Draw a rectangle around the faces
    for (x, y, w, h) in faces:
        cv2.rectangle(
            img=frame,
            pt1=(x, y),
            pt2=(x + w, y + h),
            color=(0, 255, 0),
            thickness=2,
        )

    # Display the resulting frame
    cv2.imshow("Chapter 14 - OpenCV", frame)
```

```
# Break when key "q" is pressed
if cv2.waitKey(1) == ord("q"):
    break

video_capture.release()
cv2.destroyAllWindows()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/chapter14_opencv.py

You can simply run this script by invoking it with Python:

```
$ python chapter14/chapter14_opencv.py
```

A window similar to the one shown in *Figure 14.1* will open and start streaming images from your webcam.

3. When the algorithm detects a face, it draws a green rectangle around it. Press the *q* key on your keyboard to stop the script:

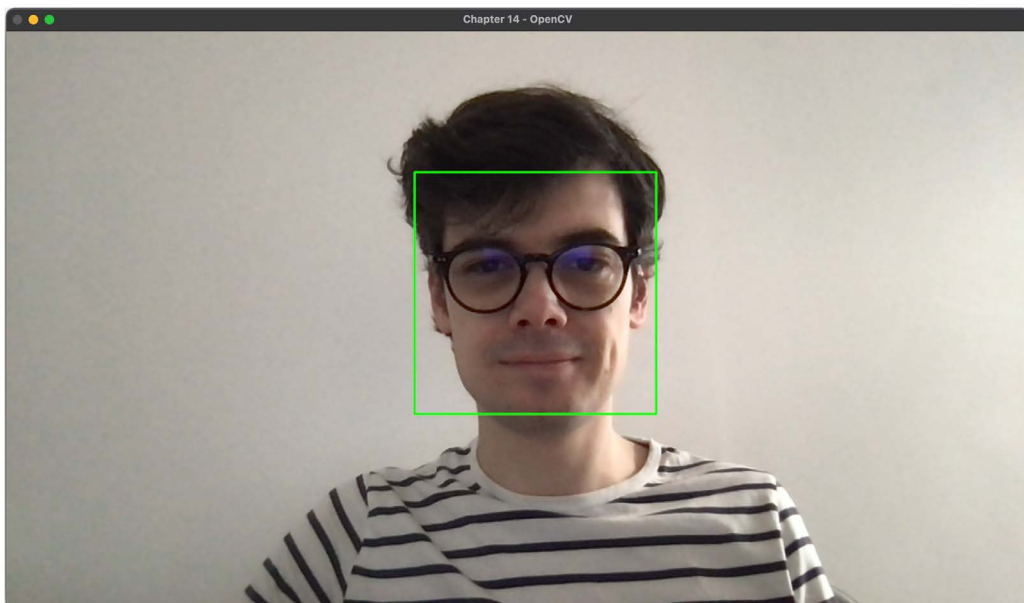


Figure 14.1 – Face detection script with OpenCV

4. Let's go through the implementation. The first thing we do is instantiate a `CascadeClassifier` class with an XML file bundled with the library. This class is actually a machine learning algorithm using the Haar cascade principle. You can read more about the theory behind this algorithm in the OpenCV documentation: https://docs.opencv.org/master/db/d28/tutorial_cascade_classifier.html.

The nice thing here is that OpenCV comes with pre-trained models, provided in the form of XML files, including ones for face detection. Hence, we only have to load them to start working on images.

5. Then, we instantiate a `VideoCapture` class. It'll allow us to stream images from a webcam. The integer argument in the initializer is the index of the camera you want to use. If you have several cameras, you may need to adjust this argument.
6. After that, we start an infinite loop so that we can continuously run detections on the stream of images. Inside it, we start by retrieving an image, `frame`, from the `video_capture` instance. This image is then fed to the classifier thanks to the `detectMultiScale` method. Notice that we first convert it to grayscale, which is a requirement for Haar cascade classifiers.

The result of this operation is a list of tuples containing the characteristics of the rectangles around the detected faces: `x` and `y` are the coordinates of the starting point; `w` and `h` are the width and height of this rectangle. All we have to do is draw each rectangle on the image using the `rectangle` function.

7. Finally, we can display the image in a window. Notice that before ending the loop, we give it a chance to break by listening for a keypress on the keyboard: if the `q` key is pressed, we break the loop.

And that's it! Fewer than 40 lines of code to have a working face detection system! As you can see, OpenCV makes our life very easy by providing trained classifiers. Besides, it comes with all the tools to capture and work on images.

Of course, our goal in this chapter is to put all this intelligence on a remote server so that we can serve this experience to thousands of users. Once again, FastAPI will be our ally here.

Implementing an HTTP endpoint to perform face detection on a single image

Before working with WebSockets, we'll start simple and implement, using FastAPI, a classic HTTP endpoint for accepting image uploads and performing face detection on them. As you'll see, the main difference from the previous example is in how we acquire the image: instead of streaming it from the webcam, we get it from a file upload that we have to convert into an OpenCV image object.

You can see the whole implementation in the following code:

chapter14_api.py

```
from typing import List, Tuple

import cv2
import numpy as np
from fastapi import FastAPI, File, UploadFile
from pydantic import BaseModel

app = FastAPI()
cascade_classifier = cv2.CascadeClassifier()

class Faces(BaseModel):
    faces: List[Tuple[int, int, int, int]]

@app.post("/face-detection", response_model=Faces)
async def face_detection(image: UploadFile = File(...)) -> Faces:
    data = np.fromfile(image.file, dtype=np.uint8)
    image = cv2.imdecode(data, cv2.IMREAD_UNCHANGED)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = cascade_classifier.detectMultiScale(gray)
    if len(faces) > 0:
        faces_output = Faces(faces=faces.tolist())
```

```
else:
    faces_output = Faces(faces=[])
    return faces_output

@app.on_event("startup")
async def startup():
    cascade_classifier.load(
        cv2.data.harcascades + "haarcascade_frontalface_
        default.xml"
    )
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/chapter14_api.py

As you can see, we start with a rather simple FastAPI application. At the top of the file, we instantiate a `CascadeClassifier` class. Notice, however, that contrary to the previous example, we load the trained model inside the startup event instead of doing it right away. This is for the same reason we explained in *Chapter 13, Creating an Efficient Prediction API Endpoint with FastAPI*, when we loaded our dumped Joblib model: we want to load it only when the application is actually starting, not when we are importing the module.

Then, we define a `face_detection` endpoint that expects `FileUpload`. If you need a refresher on file uploads, you can check out *Chapter 3, Developing a RESTful API with FastAPI*. Once we have the file, you can see that we are performing two operations using NumPy and OpenCV. Indeed, the images need to be loaded into a NumPy matrix that is usable by OpenCV.

If we had a file path, we could have directly used the `imread` function of OpenCV to load it. Here, we have an `UploadFile` object that has a `file` property pointing to a file descriptor. Using NumPy, we can load the binary data into an array of pixels, `data`. This can be used afterward by the `imdecode` function to create a proper OpenCV matrix.

Finally, we can run the prediction using the classifier, as we saw in the previous section. Notice that we structure the result into a structured Pydantic model. When OpenCV detects faces, it returns the result as a nested NumPy array. The goal of the `tolist` method is just to transform it into a standard list of lists.

You can run this example using the usual Uvicorn command:

```
$ uvicorn chapter14.chapter14_api:app
```

In the code example repository, you'll find a picture of a group of people: <https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/assets/people.jpg>.

Let's upload it on our endpoint with HTTPie:

```
$ http --form POST http://localhost:8000/face-detection  
image@./assets/people.jpg
```

```
HTTP/1.1 200 OK
```

```
content-length: 43
```

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

```
date: Wed, 21 Jul 2021 07:58:17 GMT
```

```
server: uvicorn
```

```
{
```

```
  "faces": [
```

```
    [
```

```
      237,
```

```
      92,
```

```
      80,
```

```
      80
```

```
    ],
```

```
    [
```

```
      426,
```

```
      75,
```

```
      115,
```

```
      115
```

```
    ]
```

```
  ]
```

```
}
```

The classifier was able to detect two faces in the image.

Great! Our face detection system is now available as a web server. However, our goal is still to make a real-time system: thanks to WebSockets, we'll be able to handle a stream of images.

Implementing a WebSocket to perform face detection on a stream of images

One of the main benefits of WebSockets, as we saw in *Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI*, is that it opens a full-duplex communication channel between the client and the server. Once the connection is established, messages can be passed quickly without having to go through all the steps of the HTTP protocol. Therefore, it's much more suited to sending lots of messages in real time.

The point here will be to implement a WebSocket endpoint that is able to both accept image data and run OpenCV detection on it. The main challenge here will be to handle a phenomenon known as **backpressure**. Put simply, we'll receive more images from the browser than the server is able to handle, because of the time needed to run the detection algorithm. Thus, we'll have to work with a queue (or buffer) of limited size and drop some images along the way to handle the stream in near real time.

You can read the implementation in the following sample:

app.py

```
async def receive(websocket: WebSocket, queue: asyncio.Queue):
    bytes = await websocket.receive_bytes()
    try:
        queue.put_nowait(bytes)
    except asyncio.QueueFull:
        pass

async def detect(websocket: WebSocket, queue: asyncio.Queue):
    while True:
        bytes = await queue.get()
        data = np.frombuffer(bytes, dtype=np.uint8)
        img = cv2.imdecode(data, 1)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = cascade_classifier.detectMultiScale(gray)
        if len(faces) > 0:
            faces_output = Faces(faces=faces.tolist())
        else:
            faces_output = Faces(faces=[])
```



```
        await websocket.send_json(faces_output.dict())

    @app.websocket("/face-detection")
    async def face_detection(websocket: WebSocket):
        await websocket.accept()
        queue: asyncio.Queue = asyncio.Queue(maxsize=10)
        detect_task = asyncio.create_task(detect(websocket, queue))
        try:
            while True:
                await receive(websocket, queue)
            except WebSocketDisconnect:
                detect_task.cancel()
            await websocket.close()
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/websocket_face_detection/app.py

As we said, we have two tasks: `receive` and `detect`. The first one is for reading raw bytes from the WebSocket, while the second one is for performing the detection and sending the result, exactly as we saw in the last section.

The key here is to use the `asyncio.Queue` object. This is a convenient structure allowing us to queue some data in memory and retrieve it in a **first in, first out (FIFO)** strategy. We are able to set a limit on the number of elements we store in the queue: this is how we'll be able to limit the number of images we handle.

The `receive` function is receiving data and putting it at the end of the queue. When working with `Queue`, we have two methods to put a new element in the queue: `put` and `put_nowait`. If the queue is full, the first one will wait until there is room in the queue. This is not what we want here: we want to drop images that we won't be able to handle in time. With `put_nowait`, the `QueueFull` exception is raised if the queue is full. In this case, we just pass and drop the data.

On the other hand, the `detect` function is pulling the first message from the queue and runs its detection before sending the result. In the previous section, we used the `fromfile` function to read the image data. Here, we directly have bytes data, so `frombuffer` is more appropriate.

The implementation of the WebSocket itself is a bit different from what we saw in *Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI*. Indeed, we don't want the two tasks to be concurrent here: we want to accept new images and *continuously run detections on images as they come in*.

This is why the `detect` function has its own infinite loop. By using `create_task` on this function, we schedule it in the event loop so that it starts to handle the images in the queue. Then, we have the regular WebSocket loop, which calls the `receive` function. In a sense, we could say that that `detect` runs "in the background." Notice that we ensure that this task is canceled when the WebSocket is closed so that the infinite loop is correctly stopped.

The rest of the implementation is similar to what we saw in the previous section. Our backend is now ready! Let's now see how to use its power from a browser.

Sending a stream of images from the browser in a WebSocket

In this section, we'll see how you can capture images from the webcam in the browser and send it through a WebSocket. Since it mainly involves JavaScript code, it's admittedly a bit beyond the scope of this book, but it's necessary to make the application work fully:

1. The first step is to enable a camera input in the browser, open the WebSocket connection, pick a camera image, and send it through the WebSocket. Basically, it'll work like this: thanks to the `MediaDevices` browser API, we'll be able to list all the camera inputs available on the device. With this, we'll build a selection form using which the user can select the camera they want to use. You can see the concrete JavaScript implementation in the following code:

script.js

```
window.addEventListener('DOMContentLoaded', (event) => {
  const video = document.getElementById('video');
  const canvas = document.getElementById('canvas');
  const cameraSelect = document.getElementById('camera-select');
  let socket;

  // List available cameras and fill select
  navigator.mediaDevices.enumerateDevices().then((devices) => {
    for (const device of devices) {
```

```
        if (device.kind === 'videoinput' && device.deviceId) {
            const deviceOption = document.createElement('option');
            deviceOption.value = device.deviceId;
            deviceOption.innerText = device.label;
            cameraSelect.appendChild(deviceOption);
        }
    });

    // Start face detection on the selected camera on submit
    document.getElementById('form-connect').
    addEventListener('submit', (event) => {
        event.preventDefault();

        // Close previous socket is there is one
        if (socket) {
            socket.close();
        }

        const deviceId = cameraSelect.selectedOptions[0].value;
        socket = startFaceDetection(video, canvas, deviceId);
    });
});
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/websocket_face_detection/script.js

2. Once the user submits the form, the `MediaDevices` API will allow us to start capturing video and display the output in an HTML `<video>` element. You can read all the details about the `MediaDevices` API in the MDN documentation: <https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices>.

3. In parallel, we'll also establish a connection with the WebSocket. Once it's established, we'll launch a repetitive task that captures an image from the video input and sends it to the server. To do this, we have to use a `<canvas>` element, an HTML tag dedicated to graphics drawing. It comes with a complete JavaScript API so that we can programmatically draw images in it. There, we'll be able to draw the current video image and convert it to valid JPEG bytes. If you want to know more about this, MDN gives a very detailed tutorial on `<canvas>`: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial.

The concrete JavaScript implementation of this is as follows:

script.js

```
const startFaceDetection = (video, canvas, deviceId) => {
  const socket = new WebSocket('ws://localhost:8000/face-
detection');
  let intervalId;

  // Connection opened
  socket.addEventListener('open', function () {

    // Start reading video from device
    navigator.mediaDevices.getUserMedia({
      audio: false,
      video: {
        deviceId,
        width: { max: 640 },
        height: { max: 480 },
      },
    }).then(function (stream) {
      video.srcObject = stream;
      video.play().then(() => {
        // Adapt overlay canvas size to the video size
        canvas.width = video.videoWidth;
        canvas.height = video.videoHeight;

        // Send an image in the WebSocket every 160 ms
        intervalId = setInterval(() => {
```

```
        // Create a virtual canvas to draw current video
image
        const canvas = document.createElement('canvas');
        const ctx = canvas.getContext('2d');
        canvas.width = video.videoWidth;
        canvas.height = video.videoHeight;
        ctx.drawImage(video, 0, 0);

        // Convert it to JPEG and send it to the WebSocket
        canvas.toBlob((blob) => socket.send(blob), 'image/
jpeg');
    }, IMAGE_INTERVAL_MS);
});
});
});

// Listen for messages
socket.addEventListener('message', function (event) {
    drawFaceRectangles(video, canvas, JSON.parse(event.data));
});

// Stop the interval and video reading on close
socket.addEventListener('close', function () {
    window.clearInterval(intervalId);
    video.pause();
});

return socket;
};
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/websocket_face_detection/script.js

Notice that we limit the size of the video input to 640 by 480 pixels, so that we don't blow up the server with too big images. Besides, we set the interval to run every 42 milliseconds (the value is set in the `IMAGE_INTERVAL_MS` constant), which is roughly equivalent to 24 images per second.

As you can see, we also wire the event listener to handle the messages received from the WebSocket. It calls the `drawFaceRectangles` function, which we'll detail in the next section.

Showing the face detection results in the browser

Now that we are able to send input images to the server, we have to show the result of the detection in the browser. In a similar way to what we showed in the *Getting started with OpenCV* section, we'll draw a green rectangle around the detected faces. Thus, we have to find a way to take the rectangle coordinates sent by the server and draw them in the browser:

1. To do this, we'll once again use a `<canvas>` element. This time, it'll be visible to the user and we'll draw the rectangles using it. The trick here is to use CSS positioning so that this element overlays the video: this way, the rectangles will be shown right on top of the video and the corresponding faces. You can see the HTML code here:

index.html

```
<body>
  <div class="container">
    <h1 class="my-3">Chapter 14 - Real time face detection</h1>
    <form id="form-connect">
      <div class="input-group mb-3">
        <select id="camera-select"></select>
        <button class="btn btn-success" type="submit"
id="button-start">Start</button>
      </div>
    </form>
    <div class="position-relative">
      <video id="video"></video>
      <canvas id="canvas" class="position-absolute top-0
start-0"></canvas>
    </div>
```

```
</div>  
  
<script src="script.js"></script>  
</body>
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/websocket_face_detection/index.html

The CSS class we are using is `utilities`, provided by Bootstrap, a very common CSS library. Basically, we set the canvas with absolute positioning and put it at the top left so that it covers the video element.

2. The key now is to use the Canvas API to draw the rectangles according to the received coordinates. This is the purpose of the `drawFaceRectangles` function, which is shown in the next sample code block:

script.js

```
const drawFaceRectangles = (video, canvas, faces) => {  
  const ctx = canvas.getContext('2d');  
  
  ctx.width = video.videoWidth;  
  ctx.height = video.videoHeight;  
  
  ctx.beginPath();  
  ctx.clearRect(0, 0, ctx.width, ctx.height);  
  for (const [x, y, width, height] of faces.faces) {  
    ctx.strokeStyle = "#49fb35";  
    ctx.beginPath();  
    ctx.rect(x, y, width, height);  
    ctx.stroke();  
  }  
};
```

https://github.com/PacktPublishing/Building-Data-Science-Applications-with-FastAPI/blob/main/chapter14/websocket_face_detection/script.js

With the canvas element, we can use a 2D context to draw things in the object. Notice that we first clean everything to remove the rectangles from the previous detection. Then, we simply have to loop through all the detected faces and draw a rectangle with the given `x`, `y`, `width`, and `height` values.

3. Our system is now ready and it's time to give it a try! As in *Chapter 8, Defining WebSockets for Two-Way Interactive Communication in FastAPI*, we'll start two servers: one with Uvicorn to serve the FastAPI application and another that uses the built-in Python server to serve the HTML and JavaScript files.

- In one terminal, launch the FastAPI application:

```
$ uvicorn chapter14.websocket_face_detection.app:app
```

- In another terminal, serve the HTML application with the built-in Python server:

```
$ python -m http.server --directory chapter14/websocket_face_detection 9000
```

The HTML application is now ready on port 9000. You can access it in your browser with the address `http://localhost:9000`. You'll see an interface inviting you to choose the camera you want to use, as shown in *Figure 14.2*:

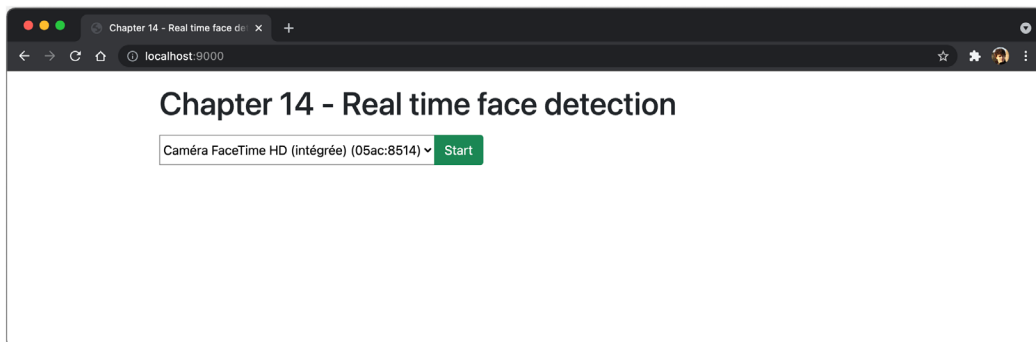


Figure 14.2 – Webcam selection for the face detection web application

4. Select the webcam you wish to use and click on **Start**. The video output will show up, face detection will start via the WebSocket and green rectangles will be drawn around the detected faces. We show this in *Figure 14.3*:

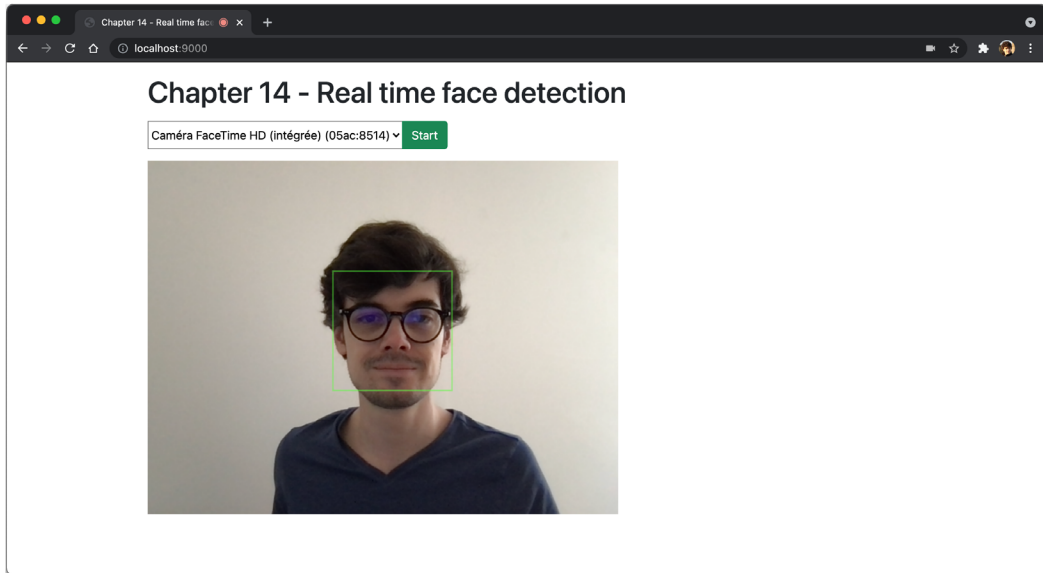


Figure 14.3 – Running the face detection web application

It works! We brought the intelligence of our Python system right into the user's web browser. This is just an example of what you could achieve using WebSockets and machine learning algorithms, but this definitely enables you to create near real-time experiences for your users.

Summary

In this chapter, we showed how WebSockets can help us bring a more interactive experience to users. Thanks to OpenCV, we were able to quickly implement a face detection system. Then, we integrated it into a WebSocket endpoint with the help of FastAPI. Finally, by using a modern JavaScript API, we sent video input and displayed algorithm results directly in the browser. All in all, a project like this might sound complex to make at first, but we saw that powerful tools such as FastAPI enable us to get results in a very short time and with very comprehensible source code.

This is the end of the book and our FastAPI journey together. We sincerely hope that you liked it and that you learned a lot along the way. We covered many subjects, sometimes just by scratching the surface, but you should now be ready to build your own projects with FastAPI and serve up smart data science algorithms. Be sure to check out all the external resources we mentioned along the way, as they will give you all the insights you need for mastery.

In recent years, Python has gained a lot of popularity, especially in the data science community, and FastAPI, even though it's still very young, is already a game-changer and has seen an unprecedented adoption rate. It'll likely be at the heart of many data science systems in the coming years... And if you've read this book, you'll probably be one of the developers behind them. Cheers!



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

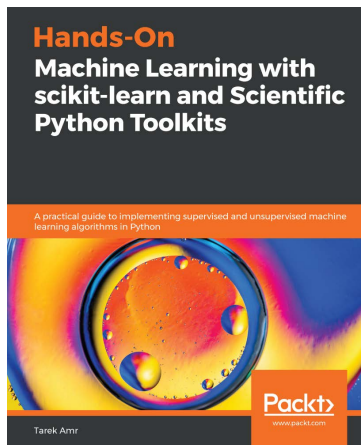
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

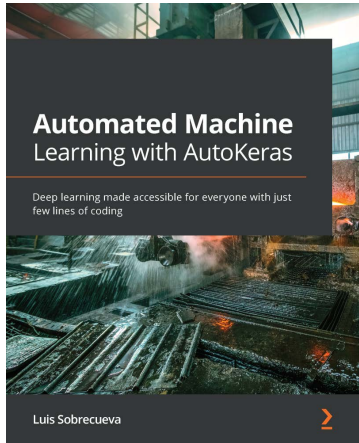


Hands-On Machine Learning with scikit-learn and Scientific Python Toolkits

Tarek Amr

ISBN: 9781838826048

- Understand when to use supervised, unsupervised, or reinforcement learning algorithms
- Find out how to collect and prepare your data for machine learning tasks
- Tackle imbalanced data and optimize your algorithm for a bias or variance tradeoff
- Apply supervised and unsupervised algorithms to overcome various machine learning challenges
- Employ best practices for tuning your algorithm's hyper parameters
- Discover how to use neural networks for classification and regression
- Build, evaluate, and deploy your machine learning solutions to production



Automated Machine Learning with AutoKeras

Luis Sobrecueva

ISBN: 9781800567641

- Set up a deep learning workstation with TensorFlow and AutoKeras
- Automate a machine learning pipeline with AutoKeras
- Create and implement image and text classifiers and regressors using AutoKeras
- Use AutoKeras to perform sentiment analysis of a text, classifying it as negative or positive
- Leverage AutoKeras to classify documents by topics
- Make the most of AutoKeras by using its most powerful extensions

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Building Data Science Applications with FastAPI*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

.env file
 creating 297
 using 296

A

Aerich
 database migration system,
 setting up 198-200
aggregating operations
 reference link 324
Alembic
 database migration system,
 setting up 180-185
Amazon ECR
 reference link 308
Amazon Elastic Container Service
 reference link 308
Amazon RDS
 reference link 303
Amazon Web Services (AWS) 254
Any, typing module 54, 55
API endpoint
 creating 62-65
 running, locally 62-65

application programming
 interface (API) 132
arguments
 accepting, with *args 30, 31
 accepting, with **kwargs 30, 31
array broadcasting
 reference link 324
arrays
 adding 322-324
 aggregating 324
 comparing 325
 creating, with NumPy 315-317
 manipulating, with NumPy 320-322
 multiplying 323, 324
async functions
 versus standard functions 369-372
asynchronous generator 205
asynchronous I/O 56-59
Asynchronous Server Gateway
 Interface (ASGI) 57
Azure App Service, CLI
 reference link 301
Azure App Service, continuous
 deployment/manual Git deployment
 reference link 302
Azure App Service, web interface
 reference link 301, 302

Azure CLI

installation link 301

Azure Database for PostgreSQL

reference link 303

B

backpressure 381

best parameters

finding, with SVM 356, 357

Boolean logic

performing 23

break statement

in Python 29

broadcasting 323

browser

face detection results,
displaying 387-390

built-in types, Python

programming 17, 18

C

Callable class

type function signatures with 53, 54

Callable object 44

camel case 39

canvas tutorial

reference link 385

cast, typing module 54, 55

C language 314

class

defining 39, 40

classification problems 334

class inheritance

used, for creating model

variations 126-128

class methods

using, as dependencies 150-152

clustering 335

computer vision 374

concurrency

handling 247-250

Conda, package managers 298

conditional statements, in Python

elif statement 26

else statement 26

executing 25-27

if statement 26

containers 304

context manager 273

continue statement

in Python 29

control flow statements

in Python 25

coroutines 57

CORS

about 228-233

configuring, in FastAPI 228-233

protecting, against CSRF

attacks 227, 228

CPython 314

cross-origin HTTP requests 228

Cross-Site Request Forgery (CSRF) 228

cross-validation 336

cryptographic hash functions 216

CSRF attacks

double-submit cookies, implementing

to prevent 233-239

CSV data

exporting 331, 332

importing 331, 332

custom data validation

adding, with pydantic 129

- custom response
 - about 106, 107
 - building 102
 - file, serving 104, 105
 - redirection, making 104
 - response_class argument, using 102, 103

D

- data
 - classifying, with Gaussian Naive Bayes 347-350
 - classifying, with multinomial Naive Bayes 350, 351
 - classifying, with Naive Bayes models 346
 - classifying, with Support Vector Machines (SVM) 351
- database
 - models and tables, creating 217, 218
 - passwords, hashing 218, 219
 - registration routes,
 - implementing 219, 220
 - user entity, storing 216
 - using, for testing 278-285
- database access token
 - endpoints, securing with 225-227
 - implementing 220, 221
 - login endpoint, implementing 222-225
- database migrations
 - managing 303
- database migration system
 - setting up, with Aerich 198-200
 - setting up, with Alembic 180-185
- database servers
 - adding 303
- dataset, loading utilities
 - reference link 337
- data structures, Python programming
 - dictionaries 21
 - lists 18, 19
 - sets 22
 - tuples 19-21
 - working with 18
- decorator 62
- deep copy 320
- dependencies
 - 404 error, raising 147, 148
 - class methods, using as 150-152
 - object, retrieving 147, 148
 - using, in WebSocket endpoints 250-252
 - using, on path decorator 153
 - using, on whole application 156, 157
 - using, on whole router 154, 155
- dependency injection
 - about 55, 142
 - advantages 143
- dependency return value 145
- dictionary
 - about 21
 - object, converting into 132-134
- DigitalOcean tutorial
 - reference link 310
- dimensionality reduction 335
- Docker
 - download and installation link 305
 - FastAPI application, deploying 304
- Dockerfile
 - about 304
 - writing, for FastAPI application 304-306
- Docker image
 - about 304
 - building 306
 - deploying 307-309
 - locally, running 307

Docker, run
 reference link 307
double submit 236
double-submit cookies
 implementing, to prevent
 CSRF attacks 233-239
dumping 360

E

email addresses
 validating, with pydantic types 124-126
endpoints
 securing, with access tokens 225-227
environment variables
 setting 292-296
 using 292-296
estimators
 about 337
 chaining, with pipelines 340-344
event loop 57

F

face detection
 results, displaying in browser 387-390
face detection, on single image
 implementing, with HTTP
 endpoint 378-380
face detection, on stream of images
 implementing, with WebSocket 381-383
FastAPI
 about 7, 142
 CORS, configuring in 228-233
 security dependencies 212-216
 used, for creating WebSocket 243-247

FastAPI application
 deploying, on serverless
 platform 300-302
 deploying, on traditional server 309
 deploying, with Docker 304
features 334
field level
 validation, applying at 129, 130
filter modifier 226
first in, first out (FIFO) 382
foreign key 163
four-space indentation 17
f-strings 16
function dependency
 creating 143-146
 using 144-146
functions
 arguments, accepting with *args 30
 arguments, accepting with **kwargs 30
 defining 29, 30

G

Gaussian Naive Bayes
 data, classifying with 347-350
gcloud CLI
 reference link 301
generator
 functions 37
 in Python 36-39
generics 50
Google App Engine, CLI
 reference link 302
Google App Engine, configuration file
 reference link 301, 302
Google Artifact Registry
 reference link 308

Google Cloud Platform (GCP) 254
Google Cloud Run
 reference link 308
Google Cloud SQL
 reference link 303
Gunicorn
 adding, as server process for
 deployment 299, 300
 reference link 309

H

Heroku
 installation link 301
Heroku, CLI
 reference link 302
Heroku, CLI/web interface
 reference link 302
Heroku, configuration file
 reference link 301
Heroku Postgres
 reference link 303
hidden files
 creating 297
holdout set 335
HTTP endpoint
 implementing, to perform face
 detection on single image 378-380
HTTP errors
 raising 99-102
HTTPIe command-line utility
 installing 9-11
HTTPX
 testing tools, setting up for
 FastAPI 270-273
HyperText Markup Language
 (HTML) 245
HyperText Transfer Protocol (HTTP) 124

I

indexing data
 reference link 328
inheritance, object-oriented programming
 logic, reusing with 45, 46
 multiple inheritance 46-48
 repetition, avoiding with 45, 46
input/output (I/O) 245
instance
 creating, from sub-class object 135, 136
 updating, with partial one 137, 138
International Organization for
 Standardization (ISO) 116
iterator 26

J

JavaScript Object Notation
 (JSON) 136, 257
Joblib
 results, caching with 366-369
 trained model, persisting with 360
join query 164
JSONResponse 102

K

kernel trick 351
keyword arguments 30

L

label 334
Linux server
 FastAPI application, deploying on 309
list comprehensions
 in Python 34-36

lists 18, 19

login endpoint

implementing 222-225

M

machine learning (ML)

about 334

model validation 335, 336

supervised, versus unsupervised
learning 334, 335

magic methods, object-oriented

programming

`__add__` operator 43

`__call__` method 44, 45

`__eq__` method 42, 43

`__gt__` method 42, 43

`__lt__` method 42, 43

`__mul__` operator 43

`__repr__` method 41, 42

`__str__` method 41, 42

`__sub__` operator 43

implementing 41

marker 266

masking 330

MediaDevices

reference link 384

message brokers 254

messages

broadcasting 253-259

Method Resolution Order (MRO) 47

Microsoft Azure Container Instances

reference link 308

Microsoft Azure Container Registry

reference link 308

mixins 47

model

creating, compatible with

MongoDB ID 201

defining, with pydantic 114

training 337-340

validating, with cross-

validation 344, 345

validating, with ML 335, 336

models, field types

defining, with pydantic 114

model variations

creating, with class inheritance 126-128

Motor, used for communicating

with MongoDB database

about 200

database, connecting to 202

documents, deleting 207, 208

documents, inserting 203, 204

documents, nesting 208-210

documents, retrieving 204-207

documents, updating 207, 208

models, creating compatible

with MongoDB ID 201

Mozilla Developer Network (MDN)

reference link 247

multi-dimensional data

pandas DataFrames, using for 329, 330

multinomial Naive Bayes

data, classifying with 350, 351

multiple inheritance 46-48

multiple WebSocket connections

handling 253-259

mypy

type checking 48

type hinting 48

N

- naïve 347
- Naive Bayes models
 - data, classifying with 346
 - intuition 346
- name collision 109
- namespace package 33
- newsgroups text dataset
 - reference link 341
- NoSQL databases
 - about 162, 164, 165
 - selecting 165
- NumPy
 - about 315
 - arrays, manipulating with 320-322
 - installing 315
 - using, to create arrays 315-317
- NumPy arrays
 - considerations 320
 - elements, accessing 318
- NumPy documentation
 - reference link 317
- NumPy user guide
 - reference link 325

O

- object
 - converting, into dictionary 132-134
- object level
 - validation, applying at 130, 131
- object-oriented programming
 - class, defining 39, 40
 - logic, reusing with inheritance 45, 46
 - magic methods, implementing 41
 - repetition, avoiding with inheritance 45, 46

- writing 39
- one-dimensional data
 - pandas Series, using for 326-328
- OpenCV
 - reference link 377
 - using 374-377
- operators, Python programming
 - about 23
 - Boolean logic, performing 23
 - value, checking in data structure 24, 25
 - variables, checking 23, 24

P

- pandas 326
- pandas DataFrames
 - using, for multi-dimensional data 328-330
- pandas Series
 - using, for one-dimensional data 326, 327
- pandas user guide
 - reference link 332
- parameterized dependency
 - creating 148
 - using 149, 150
- parametrize marker
 - used, for generating tests 265-267
- path decorator
 - dependencies, using on 153
- path operation function 62
- path operation parameters
 - used, for customizing response 88
- path parameters
 - about 65-67
 - advanced validation rules 70-72
 - values, limiting 68-70

- PennState online course
 - reference link 350, 351
- pipelines
 - about 340
 - estimators, chaining with 340-344
- Pipenv, package managers 298
- Poetry, package managers 298
- pointer 24
- POST endpoints
 - tests, writing for 276, 277
- prediction
 - running, on ML model 337-340
- prediction endpoint
 - implementing 363-366
- preflight requests 231
- pre-processors
 - chaining 340-344
- primary key 163
- projects
 - structuring, with multiple routers 107-111
- publish-subscribe (pub-sub) 254
- push
 - in Docker jargon 308
- pydantic
 - custom data validation, adding with 129
 - models, defining with 114
- pydantic data models
 - default values 120-122
 - dynamic default values 123
 - field validation 122
 - optional fields 120-122
 - standard field types 114-119
- pydantic objects
 - working with 132
- pydantic parsing
 - validation, applying before 131, 132
- pydantic types
 - used, for validating email addresses 124-126
 - used, for validating URLs 124-126
- pyenv
 - reference link 4
 - used, for installing Python distribution 4-7
- PyPi
 - URL 7
- pytest
 - unit testing 263-265
- Python 314
- Python dependencies
 - managing 297-299
- Python distribution
 - installing, with pyenv 4-7
- Python modules
 - using 31-34
 - writing 31-34
- Python packages
 - installing, with pip 8
 - using 31-34
 - writing 31-34
- Python programming
 - about 14
 - break statement 29
 - built-in types 17, 18
 - continue statement 29
 - control flow statements 25
 - data structures 18
 - generator 36-39
 - indentation 16, 17
 - list comprehensions 34-36
 - scripts, running 14, 15
 - while loop statement 28
- Python virtual environment
 - creating 7, 8

Q

query parameters 72-74

R

Radial Basis Function (RBF) 354

RandomizedSearchCV

reference link 357

Redis

about 255

reference link 255

registries 304

regression problems 334

regular expression (regex) 122

relational databases

about 162-164

selecting 165

request body

about 74-77

multiple objects 77-79

request parameters

cookies 85-87

file uploads 79-84

form data 79-81

handling 65

headers 85-87

path parameters 65-67

query parameters 72-74

request body 74-77

request object 87

response

customizing, with path

operation parameters 88

response_class argument

using 102, 103

response, customizing with path

operation parameters

response model 90-94

status code 88-90

response model 90-94

response parameter

about 95

cookies, setting 96, 97

headers, setting 95, 96

status code, setting dynamically 97-99

REST API endpoints

tests, writing for 275

results

caching, with Joblib 366-369

routers

about 107

project, structuring with 107-109, 111

S

same-origin policy 228

scikit-learn

basics 337

estimators, chaining with

pipelines 340-344

models, training 337-340

model, validating with cross-

validation 344, 345

prediction, running 337-340

pre-processors, chaining 340-344

Support Vector Machines

(SVM), using 355, 356

Secure Sockets Layer/Transport Layer

Security (SSL/TLS) 243

security dependencies

in FastAPI 212-216

- serverless platform
 - FastAPI application, deploying 300-302
- sets 22
- singular body values 78
- snake case 30
- sockets 242
- SQLAlchemy, used for communicating
 - with SQL database
 - about 166, 167
 - database, connecting to 169-171
 - database migration system, setting up with Alembic 180-185
 - delete queries, making 175-177
 - insert queries, making 171, 172
 - relationships, adding 177-180
 - select queries, making 173-175
 - table schema, creating 168, 169
 - update queries, making 175-177
- standard functions
 - versus async functions 369-372
- Starlette
 - URL 65
- static-type checkers 48
- status code 88-90
- stop words 342
- stream of images
 - sending, from browser in
 - WebSocket 383-387
- sub-arrays
 - accessing 319
- sub-class object
 - instance, creating from 135, 136
- supervised learning 334
- Support Vector Machines (SVM)
 - about 351
 - best parameters, finding 356, 357
 - data, classifying with 351
 - intuition 352-354

- using, in scikit-learn 355, 356

Support Vector Machines (SVM),
mathematical formulation
reference link 354

T

Term Frequency-Inverse Document
Frequency (TF-IDF) 342

testing

- with database 278-285

testing tools

- setting up, for FastAPI with
HTTPX 270-273

test logic

- reusing, by creating fixtures 267-270

tests

- generating, with parametrize 265-267
- writing, for POST endpoints 276, 277
- writing, for REST API endpoints 275
- writing, for WebSocket
endpoints 286-289

TF-IDF term weighting
reference link 342

Tortoise ORM, used for communicating
with SQL database

- about 186
- database migration system, setting
up with Aerich 198-200
- database models, creating 186-188
- objects, creating 190
- objects, deleting 193, 194
- objects, filtering 191, 192
- objects, retrieving 191, 192
- objects, updating 193, 194
- relationships, adding 194-197
- Tortoise engine, setting up 188, 189

trained model
 dumping 360, 361
 loading 362, 363
 persisting, with Joblib 360

train_test_split function
 reference link 339

tuples 19-21

two-way communication
 principles, with WebSockets 242

type annotations 48

type checking
 with mypy 48

type function signatures
 with Callable 53, 54

type hinting
 with mypy 48

type hinting, in Python
 working with 48-50

typing module
 about 50-53
 Any 54, 55
 cast 54, 55

U

Uniform Resource Identifiers (URIs) 243

Uniform Resource Locators (URLs)
 about 247
 validating, with pydantic types 124-126

unit testing
 with pytest 263-265

unsupervised learning 334

user identifier (UID) 257

Uvicorn 8

Uvicorn documentation
 reference link 299

V

validation
 applying, at field level 129, 130
 applying, at object level 130, 131
 applying, before pydantic
 parsing 131, 132

validators 129

W

Web Server Gateway Interface (WSGI) 56

WebSocket
 about 242
 creating, with FastAPI 243-247
 implementing, to perform face detection
 on stream of images 381-383
 stream of images, sending
 from browser 383-387
 two-way communication, principles 242

WebSocket endpoints
 tests, writing for 286-289

while loop statement
 in Python 28

whitespace indentation 16

Windows Subsystem for Linux (WSL) 245

Windows terminal application
 reference link 245

wss (WebSocket Secure) 243