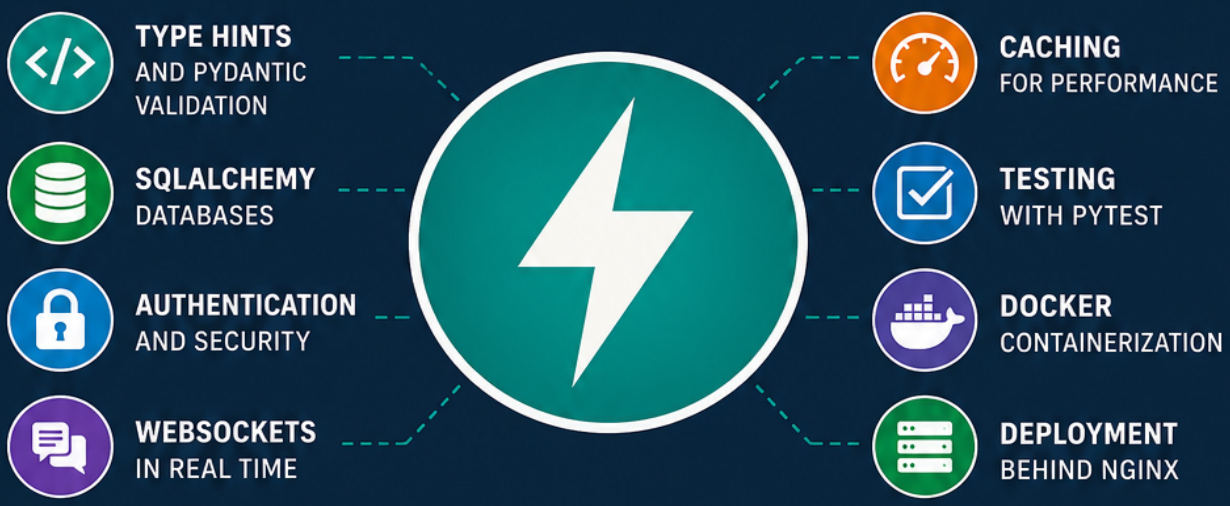
WEB DEVELOPMENT WITH FASTAPI

BUILD MODERN, PRODUCTION-READY APIS AND WEB APPLICATIONS WITH PYTHON



FROM YOUR FIRST ENDPOINT TO PRODUCTION

DESIGN • BUILD • TEST • SECURE • DEPLOY

NO PRIOR WEB FRAMEWORK EXPERIENCE REQUIRED
BEYOND BASIC PYTHON KNOWLEDGE

NICHOLAS BRADLEY

Web Development with FastAPI

Build Modern, Production-Ready APIs and Web Applications with Python

Steve Publications

This book is available at https://leanpub.com/webdevelopmentwithfastapi

This version was published on 2026-08-23

Contents

Build Modern, Production-Ready APIs and Web Applications with Python

This book takes you from your first FastAPI endpoint to designing, building, testing, securing, and deploying professional production-grade systems. You will learn the framework through a realistic application that grows chapter by chapter, gaining hands-on experience with type hints, Pydantic validation, SQLAlchemy databases, authentication, WebSockets, caching, testing with pytest, Docker containerization, and production deployment behind Nginx. No prior web framework experience is required beyond basic Python knowledge.

Introduction

You know Python. You can write functions, define classes, and import modules without thinking twice. Now you want to build web applications or APIs with it, and you have heard that FastAPI is one of the best choices available today. This book will help you find out if that is true, and show you how to put it into practice.

FastAPI has grown from a personal project by Sebastián Ramírez (known online as "Tiangolo") into one of the most popular Python web frameworks in the world. It is used by companies including Microsoft, Uber, Netflix, and Cisco for production systems handling millions of requests. Its popularity stems from three core ideas: it uses standard Python type hints to describe your API, it generates interactive documentation automatically, and it delivers performance comparable to NodeJS and Go while remaining easy to read and write.

Those are bold claims. This book will not ask you to take them on faith. Instead, we will build a complete application together, starting with a single endpoint in Chapter 1 and expanding into a fully featured system with users, authentication, databases, real-time features, caching, tests, and production deployment by the final chapter. Along the way, every concept is explained not just in terms of how to use it, but why it works that way, when you should reach for it, and when there might be a better option.

Who This Book Is For

This book assumes you are comfortable with Python basics: variables, data types, functions, classes, modules, and virtual environments. You do not need prior experience with web development frameworks, REST APIs, or asynchronous programming. If you have used Flask or Django before, many concepts will feel familiar, but FastAPI approaches them differently in important ways that we will highlight explicitly.

The book is also designed to serve as a reference once you have finished reading it. Each chapter covers its topic thoroughly with complete code examples, so you can return to specific sections when you need to implement authentication, set up database migrations, or configure production deployment.

How This Book Is Structured

The book is organized into five parts that build on each other progressively:

Part I: Foundations introduces FastAPI and the ecosystem it depends on. You will install the framework, write your first API endpoint, understand how ASGI differs from older web interfaces, and learn about Python type hints and Pydantic models that make FastAPI work.

Part II: Core Features covers the mechanisms you will use in almost every FastAPI application: routing, parameters, request bodies, responses, error handling, dependency injection, middleware, lifespan events, background tasks, configuration management, logging, and OpenAPI documentation.

Part III: Data Layer focuses on persistence with relational databases using SQLAlchemy 2.0 and async patterns. You will define models, manage relationships, handle transactions, run migrations with Alembic, and organize your code using repository and service patterns that scale to larger applications.

Part IV: Security and Advanced Features addresses authentication with OAuth2 and JWT, authorization, security best practices including OWASP API Top 10 guidance, CORS configuration, rate limiting, WebSockets for real-time communication, streaming responses, caching strategies, pagination patterns, and external API integration.

Part V: Testing, Architecture, and Production brings everything together with comprehensive testing using pytest, project structure recommendations for maintainable applications, Docker containerization, deployment with Gunicorn and Nginx, CI/CD pipelines, and observability practices for production systems.

The Running Application

Rather than presenting disconnected code snippets, this book builds one application throughout: a task management API that lets users create accounts, manage projects and tasks, invite collaborators, receive real-time updates via WebSockets, and search through their work. The application starts small and grows organically as each chapter introduces new capabilities. By the end, you will have a complete system that demonstrates how all the pieces fit together in practice.

If you want to follow along with code examples, you can build this application incrementally or jump to chapters relevant to your current needs. All code is complete and runnable with no omitted sections.

Conventions Used in This Book

Code blocks show complete files with all necessary imports included. Filenames are indicated above each code block. When a file is part of a larger project, the directory structure is shown where it first appears.

Configuration values such as secret keys, database URLs, and API tokens are shown as placeholder strings like "your-secret-key-here". Never use these placeholders in production; always generate secure random values for secrets.

Throughout the book you will encounter callout boxes with specific labels:

Note: Additional context or clarification about a concept.

Warning: A potential pitfall, security concern, or common mistake to avoid.

Best Practice: A recommendation based on industry standards and production experience.

Version Note: Information about behavior that may differ between versions of FastAPI, Pydantic, SQLAlchemy, or other libraries.

What You Will Be Able to Do After This Book

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Part I: Foundations — Understanding FastAPI and Its Ecosystem

Chapter 1: Your First FastAPI Application

The fastest way to understand what FastAPI can do is to make it work. In this chapter, you will install the framework, write your first API endpoint, run the application locally, and explore how requests flow through the system. By the end of the chapter, you will have a working API that responds to HTTP requests with JSON data, and you will understand the core concepts that every FastAPI application relies on: ASGI servers, path operations, type hints, and automatic documentation.

Installing FastAPI and Uvicorn

FastAPI is not a complete web server by itself. It is a framework that defines how your application handles requests and generates responses. To actually listen for incoming HTTP connections, you need an ASGI (Asynchronous Server Gateway Interface) server. The most common choice is Uvicorn, which is fast, reliable, and officially recommended by the FastAPI project.

Create a new directory for your project and set up a virtual environment:

```
mkdir task-api && cd task-api
python -m venv .venv
source .venv/bin/activate   # On Windows: .venv\Scripts\activate
```

Now install FastAPI with its standard extras, which include Uvicorn and other commonly used dependencies:

```
pip install "fastapi[standard]"
```

The `[standard]` extra installs several packages beyond the core framework:

- `uvicorn`: The ASGI server that runs your application

- `httpx`: An async HTTP client for making outbound requests from your API
- `jinja2`: A templating engine if you need to serve HTML pages
- `python-multipart`: Support for parsing form data and file uploads
- `itsdangerous`: Secure token generation (used in session-based authentication)

If you prefer minimal installations, you can install only what you need:

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

Verify the installation by checking the versions:

```
import fastapi
import uvicorn

print(f"FastAPI version: {fastapi.__version__}")
print(f"Uvicorn version: {uvicorn.__version__}")
```

You should see version numbers for both packages. The exact versions will depend on when you install, but FastAPI follows semantic versioning with a "0.x" series that has been stable in practice for many releases. Always pin your dependencies in production using `requirements.txt` or `pyproject.toml`.

Writing Your First API Endpoint

Create a file named `main.py` with the following content:

```
# main.py
from fastapi import FastAPI

app = FastAPI(
    title="Task Management API",
    description="A simple task management application",
    version="0.1.0"
)

@app.get("/")
async def root():
    return {"message": "Welcome to the Task Management API"}

@app.get("/health")
async def health_check():
    return {"status": "healthy"}
```

Let us break this down line by line:

- `from fastapi import FastAPI` imports the main class you use to create your application.
- `app = FastAPI(...)` creates an instance of the application. The parameters `title`, `description`, and `version` are optional but recommended because they appear in the automatically generated API documentation.
- `@app.get("/")` is a decorator that registers the function below it as a handler for HTTP GET requests to the path `/`. This is called a **path operation**. FastAPI provides similar decorators for other HTTP methods: `@app.post()`, `@app.put()`, `@app.patch()`, and `@app.delete()`.
- `async def root():` defines an asynchronous function. The `async` keyword means this function can pause while waiting for I/O operations (like database queries or external API calls) without blocking other requests. You can also use regular `def` functions; FastAPI will run them in a thread pool automatically. We discuss the trade-offs in more detail later.
- `return {"message": "Welcome to the Task Management API"}` returns a Python dictionary. FastAPI automatically converts this to a JSON response with the appropriate content type header (`application/json`).

The second endpoint, `/health`, is a simple health check that monitoring systems can call to verify your application is running. This is a standard practice for production services.

Running and Testing Your First App

Start the development server with Uvicorn:

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

This command tells Uvicorn to:

- Load the module `main` (from `main.py`)
- Use the object named `app` as your FastAPI application
- Enable `--reload`, which automatically restarts the server when you modify source files

You will see output similar to:

```
INFO:     Will watch for changes in these directories: ['/path/to/task-api']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process using WatchFiles
INFO:     Started server process [12345]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Your API is now running at `http://127.0.0.1:8000`. Open a browser and navigate to that URL. You will see the JSON response:

```
{"message": "Welcome to the Task Management API"}
```

You can also test with `curl` from another terminal:

```
curl http://127.0.0.1:8000/
# Response: {"message":"Welcome to the Task Management API"}

curl http://127.0.0.1:8000/health
# Response: {"status":"healthy"}
```

Understanding ASGI vs WSGI

Before we go further, it is important to understand what ASGI is and why FastAPI uses it instead of the older WSGI standard that frameworks like Flask and Django traditionally rely on.

WSGI (Web Server Gateway Interface) was introduced in 2003 as PEP 333. It defines a simple synchronous interface between web servers and Python applications. When a request arrives, the server calls your application function, which processes the request and returns a response. While this works well for many use cases, it has limitations:

- Each worker thread or process can handle only one request at a time
- Long-running I/O operations (database queries, external API calls) block the entire worker
- Real-time features like WebSockets are not supported natively

ASGI (Asynchronous Server Gateway Interface), defined in PEP 544 and later formalized as an independent specification, extends this model to support asynchronous code. With ASGI:

- Your application can use `async/await` to handle I/O operations without blocking
- A single worker process can manage many concurrent connections efficiently
- Long-lived connections like WebSockets and Server-Sent Events are first-class citizens
- It remains backward compatible with synchronous WSGI applications

FastAPI is built on top of Starlette, a lightweight ASGI toolkit that provides routing, middleware, and other foundational components. This means FastAPI inherits all of Starlette's async capabilities while adding its own layer for type hints, validation, and documentation.

Note: You do not need to write everything in async mode to benefit from ASGI. FastAPI will automatically run synchronous `def` functions in a thread pool so they do not block the event loop. The key advantage is that when you *do* have I/O-bound operations, you can make them truly non-blocking with `async def`.

The Request-Response Lifecycle at a Glance

Understanding how a request flows through FastAPI helps you debug problems and design better applications. Here is the high-level sequence:

1. **Transport layer**: A client (browser, curl, another service) sends an HTTP request over TCP to your server.
2. **Protocol handling**: The ASGI server (Uvicorn) parses the raw bytes into an HTTP request with method, path, headers, and body.
3. **Middleware stack**: The request passes through middleware components in reverse order of registration. Each middleware can inspect or modify the request before passing it along.
4. **Routing**: FastAPI matches the request method and path against registered path operations to find the handler function.
5. **Dependency resolution**: Before calling the handler, FastAPI resolves any dependencies declared with `Depends()`. This might include database sessions, authenticated users, or configuration values.

6. **Parameter extraction and validation**: Path parameters, query parameters, headers, cookies, and request body data are extracted from the request, validated against their type hints, and converted to the appropriate Python types.
7. **Handler execution**: Your path operation function runs with all parameters injected. It returns a value (usually a dict, Pydantic model, or custom response object).
8. **Response serialization**: FastAPI converts your return value into an HTTP response with appropriate status code, headers, and JSON body.
9. **Middleware response handling**: The response passes back through the middleware stack in forward order, allowing each to modify it if needed.
10. **Client delivery**: Uvicorn sends the final response bytes back to the client over the network.

This flow is where FastAPI's magic happens. Steps 6 and 7 are particularly important because they rely on Python type hints to automatically validate input data and provide helpful error messages when validation fails. We explore this in depth in Chapter 2.

Automatic Interactive Documentation

One of FastAPI's most distinctive features is that it generates interactive API documentation automatically, with no extra configuration required. Navigate to these URLs while your server is running:

- **Swagger UI**: `http://127.0.0.1:8000/docs` — An interactive interface where you can see all your endpoints, inspect their parameters and response schemas, and try them out directly from the browser by clicking "Try it out" and sending real requests.
- **ReDoc**: `http://127.0.0.1:8000/redoc` — An alternative documentation layout that presents a cleaner, more readable format suitable for sharing with non-technical stakeholders.

Both interfaces are generated from the OpenAPI specification that FastAPI creates automatically based on your path operations and type hints. You can view the raw OpenAPI JSON at `http://127.0.0.1:8000/openapi.json`. This specification describes your API in a standardized format that tools can consume for code generation, testing, monitoring, and more.

This automatic documentation is not a gimmick. In production environments, it serves as living documentation that stays synchronized with your code because it is generated from the same source. When you add a new endpoint or change a parameter type, the documentation updates immediately. We discuss how to customize these interfaces in Chapter 7.

Common Beginner Pitfalls

Even simple FastAPI applications can fail in subtle ways if you are not aware of common mistakes. Here are the ones beginners encounter most often:

Wrong module path when starting Uvicorn. The command `uvicorn main:app` means "load the `main` module and use the `app` object inside it." If your file is named `server.py`, you need `uvicorn server:app`. If your app is in a subdirectory like `api/main.py`, you need `uvicorn api.main:app`.

Forgetting to activate the virtual environment. If you installed FastAPI in a virtual environment but forgot to activate it before running Uvicorn, Python will not find the package and you will see a `ModuleNotFoundError`. Always verify your environment is active by checking that `.venv` appears in your shell prompt.

Using synchronous blocking code in async functions. If you write `async def my_endpoint(): time.sleep(5)`, the `time.sleep()` call blocks the entire event loop for five seconds, defeating the purpose of async code. Use `await asyncio.sleep(5)` instead for non-blocking delays. We cover this pattern in more detail in Chapter 13.

Returning unsupported types. FastAPI can automatically serialize dicts, lists, strings, numbers, booleans, Pydantic models, and some other standard types to JSON. If you return a custom class instance without defining how to serialize it, you will get an error. Either return a dict or use Pydantic models for structured responses.

Confusing path parameters with query parameters. The decorator `@app.get("/items/{item_id}")` defines a path parameter that must appear in the URL path itself (e.g., `/items/42`). Query parameters appear after a question mark (e.g., `/items?skip=0&limit=10`) and are declared as function arguments with default values or `Query()` annotations. We cover both in Chapter 3.

Summary

In this chapter, you installed FastAPI and Uvicorn, created your first API with two endpoints, ran the development server, tested it with a browser and curl, and explored the automatically generated documentation. You also learned about ASGI and how it differs from WSGI, understood the basic request-response lifecycle, and identified common pitfalls to avoid.

The application you built is simple but complete: it listens for HTTP requests, routes them to handlers based on path and method, returns JSON responses, and provides interactive documentation. Every larger FastAPI application is fundamentally this same pattern, extended with additional features like database access, authentication, validation, and background processing.

In the next chapter, we dive into Python type hints and Pydantic models, which are the foundation that makes FastAPI's automatic validation, serialization, and documentation possible. Understanding these concepts thoroughly will make everything else in this book easier to grasp.

Chapter 2: Python Type Hints and Pydantic — The Foundation of FastAPI

FastAPI is often described as "a framework built on type hints." This is not marketing language; it is a literal description of how the framework works. When you write a path operation with typed parameters, FastAPI reads those types to determine what data to extract from incoming requests, how to validate it, and what documentation to generate. If your type hints are wrong or missing, FastAPI cannot do its job correctly.

This chapter covers Python type hints and Pydantic models in detail so you understand the mechanisms behind FastAPI's validation and serialization. By the end, you will be able to define data structures that validate input automatically, convert between formats reliably, and document your API through code alone.

Python Type Hints Refresher

Python type hints were introduced in PEP 484 (2015) and have evolved significantly since then. They allow you to annotate variables, function parameters, and return values with expected types without changing runtime behavior. FastAPI relies heavily on these annotations.

Here is a quick refresher of the most important concepts:

Basic types. You can annotate variables and parameters with built-in types:

```
name: str = "Alice"
age: int = 30
price: float = 19.99
is_active: bool = True
```

In FastAPI, these annotations tell the framework what type to expect for a parameter and whether to convert it from a string (since all incoming HTTP data starts as strings).

Optional types. When a value may be absent or explicitly set to None, use `Optional`:

```
from typing import Optional

nickname: Optional[str] = None      # Can be str or None
```

In Python 3.10+, you can use the union operator instead:

```
nickname: str | None = None         # Equivalent to Optional[str]
```

Important distinction: In Pydantic v2 (which FastAPI uses), `Optional[str]` means the field is required but accepts None as a valid value. To make a field truly optional (not required in input), you must provide a default value:

```
nickname: Optional[str] = None      # Not required; defaults to None if omitted
email: str | None                    # Required in input, but can be None
```

This is a breaking change from Pydantic v1 and causes confusion for many developers migrating older code. Always include `= None` when you want an optional field that clients can omit.

Lists and sequences. Use `list` with a type parameter for homogeneous collections:

```
tags: list[str] = []                # Python 3.9+
# or
from typing import List
tags: List[str] = []                # Compatible with older Python versions
```

Dictionaries. Annotate both key and value types:

```
metadata: dict[str, str] = {}       # Keys are strings, values are strings
counts: dict[str, int] = {}         # String keys mapping to integer counts
```

Unions. When a value can be one of several types, use `Union` or the `|` operator:

```
from typing import Union

identifier: Union[int, str]         # Can be int or str
# or (Python 3.10+)
identifier: int | str               # Same meaning
```

The typing module. Python's standard library includes many useful types in the `typing` module:

- `Any`: A value of any type (bypasses type checking; use sparingly)
- `Literal["value"]`: Restricts a value to specific literal constants
- `Annotated[T, metadata]`: Attaches extra metadata to a type without changing it (critical for FastAPI's Depends and Pydantic Field)

FastAPI uses `Annotated` extensively. For example, `Annotated[str, Query(description="A search term")]` tells FastAPI that this parameter comes from the query string with a specific description. You will see this pattern throughout the book.

Pydantic Models: Definition, Validation, Serialization

Pydantic is a data validation library written in Rust and Python. It defines models using standard Python class syntax with type annotations, then validates data at runtime by checking values against those types. FastAPI uses Pydantic for all request body validation, response serialization, and configuration management.

Install Pydantic if you have not already (it comes with `fastapi[standard]`):

```
pip install pydantic
```

Here is a basic Pydantic model:

```python
from pydantic import BaseModel


class UserCreate(BaseModel):
    username: str
    email: str
    full_name: str | None = None
```

This model defines three fields:

- `username`: Required, must be a string
- `email`: Required, must be a string
- `full_name`: Optional (has default None), can be a string or None

You validate data by instantiating the model with a dictionary:

```python
# Valid input – all required fields present
user = UserCreate(
    username="alice",
    email="alice@example.com",
    full_name="Alice Smith"
)
print(user.model_dump())
# Output: {'username': 'alice', 'email': 'alice@example.com', 'full_name':
↪  'Alice Smith'}

# Invalid input – missing required field
try:
    bad_user = UserCreate(username="bob")
except Exception as e:
    print(e)
    # Output: 1 validation error for UserCreate
    # email
    #   Field required [type=missing, input_value={'username': 'bob'},
    ↪  input_type=dict]
```

Pydantic raises a `ValidationError` when validation fails. The error message includes the field name, the type of error, and the problematic input value, making it easy to debug issues both in development and when API clients receive validation errors.

Type coercion. Pydantic automatically converts values to the expected types when possible:

```
user = UserCreate(
    username="alice",
    email="alice@example.com",
    full_name=None          # Explicitly None is fine for str | None
)

# String numbers are converted to integers
class Item(BaseModel):
    quantity: int

item = Item(quantity="42")   # "42" becomes 42
print(item.quantity)         # Output: 42 (type int, not str)
```

This coercion is convenient but can hide bugs. Pydantic v2 is stricter than v1 about what it will coerce, which generally improves reliability. For example, it no longer silently converts arbitrary strings to booleans or integers with decimal parts.

Accessing fields. You access model fields like normal Python attributes:

```
user = UserCreate(username="alice", email="alice@example.com")
print(user.username)    # Output: alice
print(user.email)       # Output: alice@example.com
```

You can also iterate over fields or convert the model to a dictionary using `model_dump()`:

```
# All fields as a dict
data = user.model_dump()

# Only non-None values
data_excluded_none = user.model_dump(exclude_none=True)

# As JSON string
json_str = user.model_dump_json()
```

Note that Pydantic v2 renamed several methods from v1: `.dict()` became `.model_dump()`, `.json()` became `.model_dump_json()`, and `.parse_obj()` became `.model_validate()`. If you encounter older tutorials using the v1 names, update them to match current syntax.

Nested Models and Complex Data Structures

Real-world APIs handle nested data. Pydantic supports this naturally by allowing one model to reference another as a field type:

```
from pydantic import BaseModel, EmailStr
from typing import Optional


class Address(BaseModel):
    street: str
    city: str
    country: str
    postal_code: str


class UserCreate(BaseModel):
    username: str
    email: EmailStr
    full_name: Optional[str] = None
    address: Optional[Address] = None
```

Now you can validate nested structures in a single call:

```
user = UserCreate(
    username="alice",
    email="alice@example.com",
    address={
        "street": "123 Main St",
        "city": "Springfield",
        "country": "USA",
        "postal_code": "62701"
    }
)

print(user.address.city)   # Output: Springfield
```

Pydantic validates the nested `Address` object automatically. If any field is missing or has the wrong type, validation fails with a clear error message indicating the path to the problematic field (e.g., `address.postal_code`).

EmailStr and other specialized types. The `EmailStr` type shown above is from Pydantic's extra types package (`pydantic-extra-types`), which provides common constrained types like email addresses, URLs, phone numbers, and colors. Install it with:

```
pip install pydantic-extra-types
```

Using specialized types gives you validation for free:

```
from pydantic_extra_types.email import EmailStr

class Contact(BaseModel):
    email: EmailStr

# Valid
c = Contact(email="user@example.com")

# Invalid — malformed email
try:
    c = Contact(email="not-an-email")
except Exception as e:
    print(e)
    # Output includes: value is not a valid email address
```

Lists of models. You can have fields that are lists of nested models:

```
class Tag(BaseModel):
    name: str
    color: str = "blue"


class TaskCreate(BaseModel):
    title: str
    description: Optional[str] = None
    tags: list[Tag] = []


task = TaskCreate(
    title="Write documentation",
    tags=[
        {"name": "docs"},
        {"name": "important", "color": "red"}
    ]
)

print(len(task.tags))           # Output: 2
print(task.tags[1].color)       # Output: red
```

This pattern is exactly how you will define request bodies for your API endpoints. FastAPI reads these models to validate incoming JSON and generate the request schema shown in the interactive documentation.

Field Configuration with Field()

The `Field()` function from Pydantic lets you add constraints, defaults, descriptions, and metadata to model fields. This is essential for controlling validation behavior and enriching your API documentation.

Basic constraints on strings:

```
from pydantic import BaseModel, Field


class Username(BaseModel):
    username: str = Field(
        min_length=3,
        max_length=30,
        pattern=r"^[a-zA-Z0-9_]+$",      # Only alphanumeric and underscore
        description="User's unique username"
    )
```

The `pattern` parameter uses regular expressions. Pydantic v2 uses a Rust regex engine that is faster than Python's standard library but does not support all features (no lookarounds, for example).

Numeric constraints:

```
class Quantity(BaseModel):
    amount: float = Field(ge=0, le=10000, description="Quantity must be between
    ↪  0 and 10000")
    count: int = Field(gt=0, description="Count must be positive")
```

The constraint parameters are:

- `gt`: Greater than (exclusive)
- `ge`: Greater than or equal to (inclusive)
- `lt`: Less than (exclusive)
- `le`: Less than or equal to (inclusive)
- `multiple_of`: Value must be a multiple of this number

Default values and examples:

```python
class TaskCreate(BaseModel):
    title: str = Field(
        ...,
        min_length=1,
        max_length=200,
        description="Title of the task",
        examples=["Buy groceries"]
    )
    priority: int = Field(
        default=3,
        ge=1,
        le=5,
        description="Priority level from 1 (lowest) to 5 (highest)"
    )
```

Note the use of `...` (Ellipsis) as the first argument to `Field()` for required fields. This is equivalent to not providing a default value but makes the intent explicit when you are also specifying other Field parameters. The `examples` parameter appears in the OpenAPI documentation, helping API consumers understand what values are expected.

Aliases. Sometimes your internal field names differ from what clients should use. Aliases let you map between them:

```python
class UserCreate(BaseModel):
    full_name: str = Field(alias="fullName")


# Client sends camelCase JSON
user = UserCreate.model_validate({"fullName": "Alice Smith"})
print(user.full_name)      # Output: Alice Smith (accessed with Python name)

# When serializing, use by_alias=True to output the alias
print(user.model_dump(by_alias=True))
# Output: {'fullName': 'Alice Smith'}
```

This is useful when your frontend uses camelCase conventions while your Python code follows snake_case. You can configure this globally in Pydantic v2 using `model_config`:

```python
from pydantic import BaseModel, ConfigDict


class UserCreate(BaseModel):
    model_config = ConfigDict(
        populate_by_name=True,      # Accept both alias and field name on input
        alias_generator=lambda field_name: "".join(
            word.capitalize() if i > 0 else word
            for i, word in enumerate(field_name.split("_"))
        )                           # Auto-generate camelCase aliases
    )

    full_name: str
    date_of_birth: str


# Both work on input
user1 = UserCreate(fullName="Alice")          # Using alias
user2 = UserCreate(full_name="Bob")           # Using field name
↪  (populate_by_name=True)

# Output uses aliases by default with alias_generator
print(user1.model_dump())
# Output: {'fullName': 'Alice', 'dateOfBirth': '...'}
```

Custom Validators and Computed Fields

When built-in types and Field constraints are not enough, you can define custom validators to implement domain-specific validation rules.

Field-level validators. Use the `@field_validator` decorator to validate individual fields:

```python
from pydantic import BaseModel, field_validator, ValidationError


class UserCreate(BaseModel):
    username: str
    password: str
    password_confirm: str

    @field_validator("username")
    @classmethod
    def username_must_not_contain_spaces(cls, v: str) -> str:
        if " " in v:
            raise ValueError("Username cannot contain spaces")
```

```
        return v.lower().strip()

    @field_validator("password")
    @classmethod
    def password_must_be_strong(cls, v: str) -> str:
        if len(v) < 8:
            raise ValueError("Password must be at least 8 characters")
        if not any(c.isupper() for c in v):
            raise ValueError("Password must contain an uppercase letter")
        if not any(c.isdigit() for c in v):
            raise ValueError("Password must contain a digit")
        return v


# Valid
user = UserCreate(
    username="  Alice_Smith  ",
    password="SecurePass1",
    password_confirm="SecurePass1"
)
print(user.username)       # Output: alice_smith (lowercased and stripped)

# Invalid — weak password
try:
    bad_user = UserCreate(
        username="bob",
        password="weak",
        password_confirm="weak"
    )
except ValidationError as e:
    print(e.errors()[0]["msg"])   # Output: Value error, Password must be at
    ↪  least 8 characters
```

Key points about field validators:

- They are class methods decorated with `@field_validator("field_name")`
- They receive the raw value and must return a validated value (or raise an error)
- Raising `ValueError` converts automatically to a Pydantic validation error
- Validators run during model instantiation, before the field is assigned

Cross-field validators. When you need to validate multiple fields together, use `@model_validator`:

```python
from pydantic import BaseModel, model_validator, ValidationError


class UserCreate(BaseModel):
    password: str
    password_confirm: str

    @model_validator(mode="after")
    def passwords_must_match(self) -> "UserCreate":
        if self.password != self.password_confirm:
            raise ValueError("Passwords do not match")
        return self


# Valid
user = UserCreate(password="SecurePass1", password_confirm="SecurePass1")

# Invalid — passwords differ
try:
    bad_user = UserCreate(password="pass1", password_confirm="pass2")
except ValidationError as e:
    print(e.errors()[0]["msg"])   # Output: Value error, Passwords do not match
```

The `mode="after"` means this validator runs after all field validators have completed and after the model is instantiated. It receives `self` (the model instance) so it can access all fields. There is also `mode="before"`, which runs before field validation and receives the raw input data as a dictionary.

Computed fields. Sometimes you want to expose derived values that are not stored directly but computed from other fields. Use `@computed_field`:

```python
from pydantic import BaseModel, computed_field


class Rectangle(BaseModel):
    width: float
    height: float

    @computed_field
    @property
    def area(self) -> float:
        return self.width * self.height

    @computed_field
    @property
    def perimeter(self) -> float:
        return 2 * (self.width + self.height)

```

```
18
19  rect = Rectangle(width=10.0, height=5.0)
20  print(rect.area)           # Output: 50.0
21  print(rect.perimeter)      # Output: 30.0
22
23  # Computed fields appear in model_dump() and OpenAPI schema
24  print(rect.model_dump())
25  # Output: {'width': 10.0, 'height': 5.0, 'area': 50.0, 'perimeter': 30.0}
```

Computed fields are read-only (they cannot be set during instantiation) and appear in serialization output and API documentation. They are useful for exposing calculated values like total prices, formatted strings, or derived status indicators.

When to Use TypedDict or Dataclasses Instead of Pydantic

Pydantic is powerful but not always the right tool. Here is when you might choose alternatives:

TypedDict. Use `typing.TypedDict` when you need a simple dictionary with known keys and types, but do not need validation or serialization features:

```
1  from typing import TypedDict
2
3
4  class UserDict(TypedDict):
5      username: str
6      email: str
7      is_active: bool
8
9
10 user: UserDict = {"username": "alice", "email": "alice@example.com",
   ↪  "is_active": True}
```

TypedDict provides static type checking but no runtime validation. It is lightweight and useful for internal data structures where you trust the data source. FastAPI does not use TypedDict for request body validation because it lacks runtime checking, but you can use it internally in your application logic.

Dataclasses. Use `dataclasses` when you need mutable objects with default values and automatic `__init__`, `__repr__`, and `__eq__` methods:

```python
from dataclasses import dataclass


@dataclass
class Task:
    id: int
    title: str
    completed: bool = False
```

Dataclasses are part of the standard library and have no runtime validation overhead. However, they do not validate input types at instantiation (a string passed where an int is expected will be stored as-is). Pydantic provides this validation automatically.

Rule of thumb for FastAPI:

- Use **Pydantic models** for request bodies, response schemas, and configuration – anywhere you need runtime validation and automatic serialization
- Use **TypedDict** for simple internal dictionaries where the structure is trusted and static type hints are sufficient
- Use **dataclasses** for internal domain objects that do not interact directly with API boundaries

In practice, most FastAPI applications use Pydantic models at the API boundary (for requests and responses) and may use dataclasses or plain classes internally for business logic.

Summary

This chapter covered the foundation of FastAPI's validation and documentation capabilities: Python type hints and Pydantic models. You learned how to define models with required and optional fields, nest models within each other, apply constraints with Field(), write custom validators for complex rules, and use computed fields for derived values. You also understood when to choose TypedDict or dataclasses as alternatives.

These concepts are not just background knowledge; they are tools you will use in nearly every chapter of this book. When you define a request body for an endpoint, you write a Pydantic model. When you specify what data an endpoint

returns, you often use a Pydantic model. When you configure your application with environment variables, Pydantic Settings (a Pydantic extension) validates those values.

In the next chapter, we apply these concepts to build real API endpoints that accept path parameters, query parameters, request bodies, headers, cookies, and file uploads – all validated automatically through type hints and Pydantic models.

Part II: Core Features — Building Real APIs

Chapter 3: Routing, Parameters, and Request Bodies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Path Operations and HTTP Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Path Parameters: Typing, Validation, Regex Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Query Parameters: Optional vs Required, Defaults, Multiple Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Request Bodies with Pydantic Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Headers, Cookies, and Dependencies on Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Form Data and File Uploads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Mixing Different Parameter Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 4: Responses, Status Codes, and Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Response Models and Automatic Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

HTTP Status Codes: Choosing the Right One

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Custom Responses: JSONResponse, FileResponse, HTMLResponse, StreamingResponse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Headers and Cookies in Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Exception Handling with Custom Exception Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Global Error Handlers and Response Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 5: Dependency Injection — FastAPI's Superpower

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

What Dependency Injection Is and Why It Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Defining Dependencies with Depends()

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Dependencies on Parameters and Other Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Sub-dependencies and Dependency Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Yield-Based Dependencies for Setup and Teardown

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Dependency Override for Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Common Patterns: Database Sessions, Current User, Config Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 6: Middleware, Lifespan Events, and Background Tasks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

What Middleware Is and How It Works in ASGI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Writing Custom Middleware (Logging, Timing, Request ID)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Built-in Middleware: CORS, TrustedHost, GZip

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Lifespan Events: Startup and Shutdown Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Background Tasks for Async Work After Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

When to Use Background Tasks vs Message Queues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 7: Configuration, Logging, and OpenAPI Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Environment-Based Configuration (Pydantic Settings)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Loading Config from .env Files, Environment Variables, and Secrets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Structured Logging with Python Logging Module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Log Levels, Formatters, and Handlers for Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

OpenAPI Specification: What It Is and How FastAPI Generates It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Customizing Documentation (Swagger UI, ReDoc, API Title/Version)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Hiding Endpoints from Docs, Custom Tags, Grouping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Part III: Data Layer — Databases, ORMs, and Persistence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 8: Database Integration with SQLAlchemy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Relational Database Concepts for API Developers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Installing and Configuring SQLAlchemy with PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Async Engine and Session Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Defining Models (Declarative Base, Columns, Types)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

CRUD Operations: Create, Read, Update, Delete

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Filtering, Ordering, Pagination in Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 9: Relationships, Transactions, and Migrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

One-to-Many, Many-to-One, and Many-to-Many Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Loading Strategies: Eager vs Lazy Loading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Transaction Management in FastAPI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Database Migrations with Alembic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 10: Repository and Service Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Why Separation of Concerns Matters in APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

The Repository Pattern: Abstracting Data Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

The Service Layer: Encapsulating Business Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Wiring Repositories and Services with Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Handling Errors at Each Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

When to Skip Patterns (Small Apps) vs When They Are Essential

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Part IV: Security, Authentication, and Advanced Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 11: Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Auth vs Authorization: Key Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Password Hashing with Argon2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Session-Based Authentication (Cookies)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

JWT Tokens: Structure, Signing, Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

OAuth2 Password Flow with FastAPI Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Role-Based and Permission-Based Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Securing Individual Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 12: Security Best Practices and Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

HTTPS and TLS in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

CORS Configuration and Common Mistakes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Input Validation and Sanitization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

SQL Injection Prevention (Parameterized Queries)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Rate Limiting and Throttling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Security Headers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Secrets Management in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 13: Advanced Features — WebSockets, Streaming, Caching, Pagination, and External APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

WebSockets: Setup, Message Handling, Connection Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Server-Sent Events (SSE) as an Alternative to WebSockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Streaming Responses for Large Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Caching Strategies: In-Memory, Redis, HTTP Cache Headers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Pagination Patterns: Offset, Cursor, Keyset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Filtering and Searching APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

External API Integration with httpx

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Part V: Testing, Architecture, and Production Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 14: Testing FastAPI Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Why Test APIs and What to Test (Unit vs Integration vs E2E)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Using TestClient for Request-Level Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Testing Path Operations, Validation, and Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Mocking External Services and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Using Dependency Overrides in Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Database Testing Strategies (In-Memory SQLite, Fixtures, Transactions)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Async Test Patterns with pytest-asyncio

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Chapter 15: Architecture, Deployment, and Production Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Project Structure for Medium to Large Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Modularization with Routers and Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Docker Containerization (Multi-Stage Builds)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Production ASGI Servers: Uvicorn Workers, Gunicorn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Reverse Proxies: Nginx Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

CI/CD Pipelines (GitHub Actions Example)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Monitoring, Health Checks, and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Scaling Strategies and Performance Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/webdevelopmentwithfastapi.