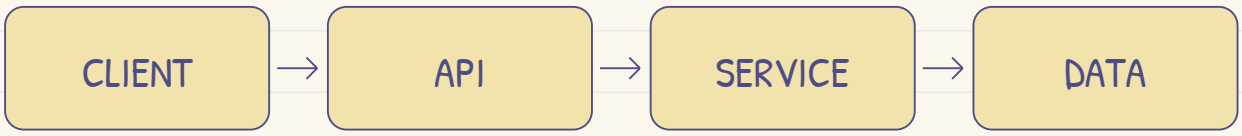
THE PRACTICAL NOTEBOOK

Python for Backend

Flask, FastAPI and dependable service design.

64 pages / 20 guided lessons / 40 code files



Understand it. Try it. Explain it.

Build JSON services with Flask and FastAPI, then learn the engineering decisions around them: validation, storage, access control, transactions, retries, testing and deployment. Small examples keep the concepts visible. The final task-service project connects the pieces without pretending that a classroom demo is a production platform.

PRIVATUS COSMAS / PRIVER

depriver.tech · First edition / October 2026

Your 20-lesson route

Work in order, or use this clickable map to revisit a topic. Each lesson has a learn page, a practice page and an explained solution.

01	Design an API contract	4
02	Flask JSON endpoints	7
03	FastAPI and typed paths	10
04	Request models	13
05	Response models	16
06	Separate service logic	19
07	Persistence and SQL	22
08	Transactions and rollback	25
09	Authentication boundaries	28
10	Authorisation and ownership	31
11	Pagination and filters	34
12	Idempotency and retries	37
13	Dependencies and test doubles	40
14	Async without confusion	43
15	Background jobs	46
16	Caching and invalidation	49
17	Logs and error handling	52
18	Configuration and releases	55
19	Tests at different boundaries	58
20	Project: a task API	61

Setup and study method · 3 References and next steps · 64

Make this book yours

BEFORE YOU START

Know Python functions, dictionaries and exceptions. Basic HTTP knowledge helps; the first lessons explain the service contract. The Python for Web book covers HTML websites, while this book focuses on machine-readable interfaces and backend behaviour.

YOUR WORKSPACE

Create a virtual environment using python -m venv .venv. Activate it with source .venv/bin/activate on macOS/Linux or .venv\Scripts\activate.bat in Windows Command Prompt. Install the packages below. Run each file independently; framework examples include in-process requests. Never deploy hardcoded demonstration credentials.

TERMINAL / FOLLOW THE COMMANDS FOR THIS BOOK

```
python -m pip install Flask "fastapi[standard]" httpx
python lesson_01_example.py
python lesson_01_solution.py
```

A SIMPLE STUDY RHYTHM

1. Read: predict what the example will do before running it.
2. Practise: write your answer using the acceptance cases.
3. Compare: read the solution, explain the difference, then extend it.

The companion ZIP contains lesson_01_example.py through lesson_20_example.py and matching solution files. Run them individually from the extracted folder. Most successful checks are silent; a traceback means a check failed.

READABLE NOTES, PRECISE CODE

Handwritten headings keep the notebook feel; body text and code use clear fonts. Copy runnable code from the ZIP to preserve indentation. The PDF is searchable and its contents links are clickable.

Design an API contract

THE GOAL

State what a client sends and what it receives.

An API contract describes paths, methods, input shapes, responses and failures. Start with the user task before choosing framework syntax. GET reads; POST commonly creates or triggers work; PUT replaces a representation; PATCH changes selected fields; DELETE removes a resource. Exact semantics must be documented and tested.

lesson_01_example.py

```
contract = {
    "GET /tasks": {"status": 200, "body": {"items": []}},
    "POST /tasks": {"status": 201, "required": ["title"]},
    "GET /tasks/unknown": {"status": 404},
}
assert contract["POST /tasks"]["status"] == 201
assert "title" in contract["POST /tasks"]["required"]
```

This executable dictionary is a design note, not a running server. It forces you to choose response behaviour before implementation details obscure it.

WATCH FOR THIS

Changing field names without coordinating clients is an API change even when the server still runs.

Your turn: Design an API contract

THE CHALLENGE

Define response envelopes for a successful item and a missing item. Keep the public error message separate from internal debugging information.

ACCEPTANCE CHECKS

[ ] Success contains id and title

[ ] Missing item has a stable error code

[ ] No traceback or credential in the error

A NUDGE, IF YOU NEED ONE

Use a small error object whose keys remain consistent across routes.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DESIGN AN API CONTRACT

lesson_01_solution.py

```
def success(ident, title):
    return {"item": {"id": ident, "title": title}}
def missing():
    return {"error": {"code": "not_found",
                      "message": "Task not found"}}
assert success(1, "Learn")["item"]["id"] == 1
assert missing()["error"]["code"] == "not_found"
assert "traceback" not in str(missing()).lower()
```

WHY IT WORKS

A stable shape makes client code simpler. The HTTP status is still needed; an error envelope returned with 200 can mislead clients and monitoring.

TAKE IT FURTHER

Write three request-and-response examples for a future task API, including one validation failure.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Flask JSON endpoints

THE GOAL

Return structured data with explicit status codes.

Flask can serialise dictionaries as JSON. Keep route functions small: read input, call domain logic and build a response. A JSON API does not need HTML templates. The response content type matters because clients need to know how to parse the body.

lesson_02_example.py

```
from flask import Flask
app = Flask(__name__)
@app.get("/health")
def health():
    return {"status": "ok"}
c = app.test_client()
r = c.get("/health")
assert r.status_code == 200
assert r.is_json
assert r.json == {"status": "ok"}
```

The test checks status, format and payload. A route returning the string representation of a dictionary would not be equivalent JSON.

WATCH FOR THIS

Flask returning JSON does not automatically validate every incoming payload.

Your turn: Flask JSON endpoints

THE CHALLENGE

Add a read-only item endpoint with a 404 response for an unknown ID. Keep error bodies JSON too.

ACCEPTANCE CHECKS

[ ] Known ID -> 200 JSON

[ ] Missing ID -> 404 JSON

[ ] No HTML error page for this handled failure

A NUDGE, IF YOU NEED ONE

Return a tuple containing the response dictionary and status.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

FLASK JSON ENDPOINTS

lesson_02_solution.py

```
from flask import Flask
app = Flask(__name__)
@app.get("/items/<int:ident>")
def item(ident):
    if ident != 1:
        return {"error": "not_found"}, 404
    return {"id": 1, "title": "Notebook"}
c = app.test_client()
assert c.get("/items/1").json["title"] == "Notebook"
r = c.get("/items/99")
assert r.status_code == 404 and r.is_json
```

WHY IT WORKS

This route handles its own missing-record case. A complete API also needs a consistent policy for routing failures and unexpected errors.

TAKE IT FURTHER

Add a list endpoint and test that its empty state is an empty array inside a stable envelope.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

FastAPI and typed paths

THE GOAL

Use declared types to validate request structure.

FastAPI reads type annotations and declarations to validate inputs and generate an OpenAPI description. A type declaration communicates a contract, but domain rules still need explicit checks. Path parsing errors and missing database records are different failures.

lesson_03_example.py

```
from fastapi import FastAPI
from fastapi.testclient import TestClient
app = FastAPI()
@app.get("/double/{value}")
def double(value: int):
    return {"result": value * 2}
c = TestClient(app)
assert c.get("/double/3").json() == {"result": 6}
assert c.get("/double/abc").status_code == 422
```

The framework rejects a value that cannot be parsed as an integer before the handler runs. Test the behaviour that your clients will see rather than assuming all bad inputs become 400.

WATCH FOR THIS

Type hints on an ordinary Python function do not validate its arguments by themselves.

Your turn: FastAPI and typed paths

THE CHALLENGE

Create an endpoint that squares a non-negative path value. Return 400 for a negative integer and allow zero.

ACCEPTANCE CHECKS

[ ] /square/4 -> 16

[ ] /square/0 -> 0

[ ] /square/-1 -> 400

[ ] /square/text -> 422

A NUDGE, IF YOU NEED ONE

Raise HTTPException for the domain rule after successful type parsing.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

FASTAPI AND TYPED PATHS

lesson_03_solution.py

```
from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
app = FastAPI()
@app.get("/square/{value}")
def square(value: int):
    if value < 0:
        raise HTTPException(400, "Use a non-negative value")
    return {"result": value * value}
c = TestClient(app)
assert c.get("/square/4").json()["result"] == 16
assert c.get("/square/0").json()["result"] == 0
assert c.get("/square/-1").status_code == 400
assert c.get("/square/text").status_code == 422
```

WHY IT WORKS

Framework validation checks representation; your function checks meaning. Keep those responsibilities clear when documenting errors.

TAKE IT FURTHER

Inspect /openapi.json and identify the declared path parameter and response operations.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Request models

THE GOAL

Validate a document before modifying state.

Pydantic models describe structured request bodies. Declare bounds and field types close to the API boundary. Validation should happen before database writes or external calls. Decide whether unknown fields are accepted or rejected so clients do not silently send misspelled properties.

lesson_04_example.py

```
from pydantic import BaseModel, Field, ConfigDict
class TaskInput(BaseModel):
    model_config = ConfigDict(extra="forbid")
    title: str = Field(min_length=2, max_length=80)
    priority: int = Field(ge=1, le=5)

task = TaskInput(title="Learn", priority=2)
assert task.priority == 2
assert task.model_dump()["title"] == "Learn"
```

The model constrains length and numeric range. A whitespace-only title can still pass a length test, so normalisation and domain validation remain important.

WATCH FOR THIS

Coercion may accept values you expected to reject. Use strict types when that matches the contract and test examples explicitly.

Your turn: Request models

THE CHALLENGE

Create a model that strips whitespace and rejects a blank title. Add tests for a valid title, blank input and an unknown field.

ACCEPTANCE CHECKS

[ ] " Learn " -> "Learn"

[ ] Spaces only -> validation error

[ ] Extra field -> validation error

A NUDGE, IF YOU NEED ONE

Use str_strip_whitespace in model configuration plus min_length.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

REQUEST MODELS

lesson_04_solution.py

```
from pydantic import BaseModel, Field, ConfigDict, ValidationError
class TaskInput(BaseModel):
    model_config = ConfigDict(extra="forbid", str_strip_whitespace=True)
    title: str = Field(min_length=1, max_length=80)
assert TaskInput(title="  Learn  ").title == "Learn"
for data in [{"title": "   "}, {"title": "A", "unknown": 1}]:
    try:
        TaskInput(**data)
    except ValidationError:
        pass
    else:
        raise AssertionError("Invalid payload accepted")
```

WHY IT WORKS

The boundary converts accepted input into a predictable model. Tests demonstrate both normalisation and rejection, which are different promises.

TAKE IT FURTHER

Add an optional due date and reject dates that violate a clearly stated business rule.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Response models

THE GOAL

Control which fields cross the service boundary.

Internal records may contain private fields or implementation details. A response model defines the public view and reduces accidental exposure. Do not return an entire user or database object merely because serialisation is convenient. Add fields deliberately and document their meaning.

lesson_05_example.py

```
from fastapi import FastAPI
from fastapi.testclient import TestClient
from pydantic import BaseModel
class PublicUser(BaseModel):
    id: int
    name: str
app = FastAPI()
@app.get("/me", response_model=PublicUser)
def me():
    return {"id": 1, "name": "Asha", "secret": "internal"}
r = TestClient(app).get("/me")
assert r.json() == {"id": 1, "name": "Asha"}
```

The response model excludes the secret field from this response. Authorisation still decides whether the caller should see even the public fields.

WATCH FOR THIS

Redacting only in the UI is too late: the network response has already exposed the data.

Your turn: Response models

THE CHALLENGE

Define a public task response that exposes id, title and done, while excluding owner_token from an internal record.

ACCEPTANCE CHECKS

[ ] Public fields are present

[ ] owner_token is absent

[ ] JSON done is a boolean

A NUDGE, IF YOU NEED ONE

Use a separate model instead of reusing a database model with every column.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

RESPONSE MODELS

lesson_05_solution.py

```
from pydantic import BaseModel
class PublicTask(BaseModel):
    id: int
    title: str
    done: bool
internal = {"id": 1, "title": "Read", "done": False,
            "owner_token": "private"}
public = PublicTask(**internal).model_dump()
assert public == {"id": 1, "title": "Read", "done": False}
assert "owner_token" not in public
```

WHY IT WORKS

Separate input, internal and output shapes as the application grows. A field allowed in a response is not necessarily editable by the caller.

TAKE IT FURTHER

Write a regression test that fails if a new internal credential field appears in API output.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Separate service logic

THE GOAL

Keep business rules testable without HTTP.

A route should translate transport details, while a service function enforces domain rules. This keeps validation consistent across HTTP, command-line tools and background jobs. Avoid hiding every decision inside decorators or database queries.

lesson_06_example.py

```
def create_task(title, priority):
    title = title.strip()
    if not title:
        raise ValueError("Title required")
    if priority not in range(1, 6):
        raise ValueError("Priority must be 1-5")
    return {"title": title, "priority": priority, "done": False}
assert create_task("  Learn  ", 2)["title"] == "Learn"
```

The function knows nothing about Flask or FastAPI. A route can convert ValueError into the agreed HTTP error without duplicating the underlying rule.

WATCH FOR THIS

Framework-neutral code can still have hidden dependencies if it reads globals or environment variables internally.

Your turn: Separate service logic

THE CHALLENGE

Implement a transition that completes a task and rejects an attempt to complete it twice. Return a new record instead of changing the original.

ACCEPTANCE CHECKS

[ ] Incomplete -> completed copy

[ ] Original remains incomplete

[ ] Already completed -> ValueError

A NUDGE, IF YOU NEED ONE

Check the rule first, then copy with a dictionary expansion.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SEPARATE SERVICE LOGIC

lesson_06_solution.py

```
def complete(task):
    if task["done"]:
        raise ValueError("Already completed")
    return {**task, "done": True}
original = {"id": 1, "done": False}
updated = complete(original)
assert updated["done"] and not original["done"]
try:
    complete(updated)
except ValueError:
    pass
else:
    raise AssertionError("Repeated transition accepted")
```

WHY IT WORKS

This is one chosen domain policy. Another API might deliberately make completing an already-complete task idempotent; choose and document it.

TAKE IT FURTHER

Pass a clock function into a service that records completion time so tests can use a fixed timestamp.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Persistence and SQL

THE GOAL

Use database constraints to support application rules.

A database stores durable state and can enforce uniqueness and required values. Application checks improve messages, while database constraints protect integrity under concurrency. Parameterised SQL handles values safely. Keep connection lifetime and transaction boundaries explicit.

lesson_07_example.py

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE tasks (id INTEGER PRIMARY KEY, "
            "title TEXT NOT NULL UNIQUE)")
con.execute("INSERT INTO tasks(title) VALUES (?)", ("Read",))
con.commit()
assert con.execute("SELECT title FROM tasks").fetchone() == ("Read",)
con.close()
```

The primary key identifies rows and the uniqueness constraint rejects duplicate titles in this specific design. Real task systems may allow repeated titles and use a different uniqueness rule.

WATCH FOR THIS

An in-memory database is a teaching fixture, not durable storage across process restarts.

Your turn: Persistence and SQL

THE CHALLENGE

Insert a duplicate title, catch the integrity error and confirm that there is still only one stored row.

ACCEPTANCE CHECKS

[ ] First insert succeeds

[ ] Duplicate insert raises IntegrityError

[ ] Count remains one

A NUDGE, IF YOU NEED ONE

Catch sqlite3.IntegrityError and roll back the failed transaction.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PERSISTENCE AND SQL

lesson_07_solution.py

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE tasks(title TEXT UNIQUE)")
con.execute("INSERT INTO tasks VALUES (?)", ("Read",))
con.commit()
try:
    con.execute("INSERT INTO tasks VALUES (?)", ("Read",))
    con.commit()
except sqlite3.IntegrityError:
    con.rollback()
assert con.execute("SELECT COUNT(*) FROM tasks").fetchone()[0] == 1
con.close()
```

WHY IT WORKS

Integrity rules belong in storage as well as application code because two requests can pass an existence check at the same time.

TAKE IT FURTHER

Move this example to a temporary file and reopen the connection to prove committed data persists.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Transactions and rollback

THE GOAL

Keep related writes all-or-nothing.

A transaction groups changes that must succeed together. If one step fails, rollback prevents a partially updated state. Keep transactions short and avoid waiting for external services while holding locks. A remote API call and a database transaction do not automatically form one atomic operation.

lesson_08_example.py

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE stock(id INTEGER PRIMARY KEY, qty INTEGER)")
con.execute("INSERT INTO stock VALUES (1, 5)")
con.commit()
try:
    with con:
        con.execute("UPDATE stock SET qty = qty - 1 WHERE id = 1")
        raise RuntimeError("Simulated failure")
except RuntimeError:
    pass
assert con.execute("SELECT qty FROM stock").fetchone()[0] == 5
con.close()
```

The connection context rolls back when the exception escapes the block. Closing the connection is still a separate responsibility.

WATCH FOR THIS

Catching an exception inside the transaction and suppressing it can accidentally allow a commit unless you roll back deliberately.

Your turn: Transactions and rollback

THE CHALLENGE

Use a conditional update to reserve one unit only when stock is positive. Verify the second reservation fails when only one unit existed.

ACCEPTANCE CHECKS

[ ] Initial stock 1 -> first reservation succeeds

[ ] Second reservation -> no row changed

[ ] Final stock is 0, never negative

A NUDGE, IF YOU NEED ONE

Put qty > 0 in the UPDATE condition and inspect rowcount.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TRANSACTIONS AND ROLLBACK

lesson_08_solution.py

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE stock(id INTEGER PRIMARY KEY, qty INTEGER)")
con.execute("INSERT INTO stock VALUES (1, 1)")
con.commit()
def reserve():
    with con:
        result = con.execute("UPDATE stock SET qty=qty-1 "
                             "WHERE id=1 AND qty>0")
        return result.rowcount == 1
assert reserve()
assert not reserve()
assert con.execute("SELECT qty FROM stock").fetchone()[0] == 0
con.close()
```

WHY IT WORKS

The condition and update happen in one SQL statement, avoiding a separate read-then-write gap. Wider workflows may need stronger isolation or locking policies.

TAKE IT FURTHER

Add a reservation record in the same transaction and prove that failure of either write restores the previous state.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Authentication boundaries

THE GOAL

Verify who is calling before doing protected work.

Authentication establishes a caller identity. A simple service-to-service API key is different from a complete user login system. Store real secrets outside source code, compare carefully, rotate them and require HTTPS. User passwords should use a maintained password-hashing implementation rather than a home-made hash.

lesson_09_example.py

```
from hmac import compare_digest

def authenticated(expected, supplied):
    return bool(expected and supplied) and compare_digest(
        expected, supplied)
assert authenticated("local-demo-key", "local-demo-key")
assert not authenticated("local-demo-key", "wrong")
assert not authenticated("", "")
```

This helper demonstrates a credential comparison, not a full authentication platform. It has no rate limiting, expiry, audit trail or user account recovery.

WATCH FOR THIS

Putting an API key into browser JavaScript makes it available to every visitor; it cannot protect a private owner API.

Your turn: Authentication boundaries

THE CHALLENGE

Create a FastAPI dependency that rejects a missing or incorrect X-API-Key. Use a fixed key only inside this isolated test.

ACCEPTANCE CHECKS

[ ] Missing key -> 401

[ ] Wrong key -> 401

[ ] Correct demonstration key -> 200

A NUDGE, IF YOU NEED ONE

Use Header and Depends so protected endpoints share the check.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

AUTHENTICATION BOUNDARIES

lesson_09_solution.py

```
from fastapi import FastAPI, Header, Depends, HTTPException
from fastapi.testclient import TestClient
from hmac import compare_digest
app = FastAPI()
def owner(x_api_key: str = Header(default="")):
    if not compare_digest(x_api_key, "local-demo-key"):
        raise HTTPException(401, "Authentication required")
@app.get("/private", dependencies=[Depends(owner)])
def private():
    return {"ok": True}
c = TestClient(app)
assert c.get("/private").status_code == 401
assert c.get("/private", headers={"X-API-Key": "wrong"}).status_code == 401
assert c.get("/private", headers={
    "X-API-Key": "local-demo-key"}).status_code == 200
```

WHY IT WORKS

In deployment, read the expected key from a secret store and fail closed if it is missing. The demo constant is intentionally not a real credential.

TAKE IT FURTHER

Add credential rotation and tests proving both the missing-secret and wrong-secret paths deny access.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Authorisation and ownership

THE GOAL

Check what an authenticated caller may do.

Authorisation is separate from authentication. A valid user must not gain access to another user's record just by guessing its ID. Check ownership or permissions at the service boundary on every protected operation. Hiding a button in the UI is not an access-control check.

lesson_10_example.py

```
def can_read(user_id, task):
    return task["owner_id"] == user_id

task = {"id": 9, "owner_id": 2, "title": "Private"}
assert can_read(2, task)
assert not can_read(3, task)
```

The user identifier must come from trusted authentication, not a body field supplied by the caller.

WATCH FOR THIS

A request body containing owner_id is not proof of ownership. Derive it from the authenticated principal.

Your turn: Authorisation and ownership

THE CHALLENGE

Return a task only when its owner matches the authenticated user. Use the same not-found response for absent and inaccessible records in this exercise.

ACCEPTANCE CHECKS

[ ] Owner gets the task

[ ] Other user receives None

[ ] Missing ID receives None

A NUDGE, IF YOU NEED ONE

Look up the record, then compare ownership before returning it.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

AUTHORISATION AND OWNERSHIP

lesson_10_solution.py

```
def find_owned(tasks, ident, user_id):
    task = tasks.get(ident)
    if task is None or task["owner_id"] != user_id:
        return None
    return task

tasks = {1: {"id": 1, "owner_id": 7, "title": "Read"}}
assert find_owned(tasks, 1, 7)["title"] == "Read"
assert find_owned(tasks, 1, 8) is None
assert find_owned(tasks, 99, 7) is None
```

WHY IT WORKS

Returning the same public result for absence and denied ownership can reduce information leakage. Internal logs can still record the actual reason without revealing private data to the caller.

TAKE IT FURTHER

Add a shared-read permission and test that it does not accidentally grant write access.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Pagination and filters

THE GOAL

Bound work per request and return stable results.

An API that returns every record can become slow or exhaust memory. Validate limits, establish stable ordering and document pagination. Offset pagination is simple; cursor pagination can use a unique increasing key to continue from the last returned record.

lesson_11_example.py

```
def after_id(rows, cursor=0, limit=2):
    if not 1 <= limit <= 100:
        raise ValueError("Limit must be 1-100")
    ordered = sorted(rows, key=lambda row: row["id"])
    return [row for row in ordered if row["id"] > cursor][:limit]
rows = [{"id": 3}, {"id": 1}, {"id": 2}]
assert after_id(rows, 1) == [{"id": 2}, {"id": 3}]
```

This list-based example teaches cursor meaning. A database implementation should push filtering, ordering and limit into SQL rather than loading the full table.

WATCH FOR THIS

A cursor should represent the chosen ordering, not an unrelated count or page label.

Your turn: Pagination and filters

THE CHALLENGE

Return an items envelope and next_cursor only when more rows remain. Read one extra matching row to decide whether there is another page.

ACCEPTANCE CHECKS

[ ] Three rows, limit 2 -> cursor 2

[ ] Last page -> cursor None

[ ] Empty input -> empty items and None

A NUDGE, IF YOU NEED ONE

Slice limit + 1 rows, return only limit, and use the last returned ID when an extra row exists.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PAGINATION AND FILTERS

lesson_11_solution.py

```
def page(rows, cursor=0, limit=2):
    if not 1 <= limit <= 100:
        raise ValueError("Invalid limit")
    found = [r for r in sorted(rows, key=lambda r: r["id"])
             if r["id"] > cursor][:limit+1]
    items = found[:limit]
    nxt = items[-1]["id"] if len(found) > limit else None
    return {"items": items, "next_cursor": nxt}
rows = [{"id": 1}, {"id": 2}, {"id": 3}]
assert page(rows)["next_cursor"] == 2
assert page(rows, 2)["next_cursor"] is None
assert page([])["items"] == []
```

WHY IT WORKS

A unique key makes the continuation unambiguous. If ordering uses a non-unique timestamp, include an ID tie-breaker in the cursor.

TAKE IT FURTHER

Add a filter and explain whether changing it invalidates an existing cursor.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Idempotency and retries

THE GOAL

Make repeated requests safe when a response is lost.

A client may retry because it never received the response, even though the server completed the action. Idempotency keys let the server recognise a repeated logical request. Store the key, request identity and result atomically in durable storage. A simple in-memory example only demonstrates the idea.

lesson_12_example.py

```
results = {}
def create_once(key, title):
    if key in results:
        old_title, result = results[key]
        if old_title != title:
            raise ValueError("Key reused with different input")
        return result
    result = {"id": len(results) + 1, "title": title}
    results[key] = (title, result)
    return result
assert create_once("abc", "Read") == create_once("abc", "Read")
assert len(results) == 1
```

The same key and input reuse the result; the same key with different input is rejected. A real implementation must also handle two requests arriving at once.

WATCH FOR THIS

Do not retry every exception: invalid input will not improve after waiting.

Your turn: Idempotency and retries

THE CHALLENGE

Write a retry wrapper that retries only a declared transient exception and gives up after a fixed attempt count. Do not sleep in this test.

ACCEPTANCE CHECKS

[ ] Immediate success -> one call

[ ] Two transient failures -> third call succeeds

[ ] Permanent exception -> no retry

A NUDGE, IF YOU NEED ONE

Catch only the transient exception and re-raise on the last attempt.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

IDEMPOTENCY AND RETRIES

lesson_12_solution.py

```
def retry(operation, attempts=3):
    if attempts < 1:
        raise ValueError("At least one attempt")
    for attempt in range(attempts):
        try:
            return operation()
        except TimeoutError:
            if attempt == attempts - 1:
                raise
calls = []
def flaky():
    calls.append(1)
    if len(calls) < 3:
        raise TimeoutError()
    return "ok"
assert retry(flaky) == "ok"
assert len(calls) == 3
```

WHY IT WORKS

Production retries also need backoff, jitter and a total deadline. Retrying a non-idempotent action without a server-side policy can duplicate work.

TAKE IT FURTHER

Test a failing operation that always times out and confirm exactly the configured number of attempts.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Dependencies and test doubles

THE GOAL

Replace external dependencies without rewriting handlers.

Dependency injection supplies a collaborator instead of creating it deep inside a function. It makes behaviour easier to test with a fixed clock, fake repository or controlled HTTP client. FastAPI offers dependency overrides; ordinary Python functions can simply accept arguments.

lesson_13_example.py

```
def report(clock):
    return {"generated_at": clock(), "status": "ok"}
assert report(lambda: "2026-10-19T00:00:00Z") == {
    "generated_at": "2026-10-19T00:00:00Z", "status": "ok"}
```

The test is deterministic because time is an input. Hidden reads of the current time would make exact output harder to reproduce.

WATCH FOR THIS

A test double that always returns ideal data can hide errors in the real integration contract.

Your turn: Dependencies and test doubles

THE CHALLENGE

Create a FastAPI route with an injected repository function, then override it in a test to return a controlled record.

ACCEPTANCE CHECKS

[ ] Override returns the test record

[ ] Route uses the dependency rather than a global lookup

[ ] Override can be cleared after the test

A NUDGE, IF YOU NEED ONE

Assign the replacement callable to app.dependency_overrides.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DEPENDENCIES AND TEST DOUBLES

lesson_13_solution.py

```
from fastapi import FastAPI, Depends
from fastapi.testclient import TestClient
app = FastAPI()
def repository():
    return {"title": "Default"}
@app.get("/task")
def task(record=Depends(repository)):
    return record
app.dependency_overrides[repository] = lambda: {"title": "Test"}
assert TestClient(app).get("/task").json() == {"title": "Test"}
app.dependency_overrides.clear()
```

WHY IT WORKS

The route is tested against a controlled dependency. Separate integration tests are still needed for the real database or external service.

TAKE IT FURTHER

Add tests where the repository reports not-found and where it raises an expected availability error.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Async without confusion

THE GOAL

Distinguish waiting from CPU-heavy computation.

Async code lets one task yield while waiting for another operation. It does not make expensive Python calculations automatically faster. Use await with non-blocking clients; calling blocking code inside an async handler can block the event loop. Synchronous handlers may be the clearer choice for synchronous libraries.

lesson_14_example.py

```
import asyncio
async def fetch_label(label):
    await asyncio.sleep(0)
    return label.upper()
async def main():
    return await asyncio.gather(fetch_label("a"), fetch_label("b"))
assert asyncio.run(main()) == ["A", "B"]
```

The zero-duration sleep yields control without introducing a real delay. gather collects results in input order, not completion order.

WATCH FOR THIS

Creating thousands of unbounded tasks can overload a service even if every call is non-blocking.

Your turn: Async without confusion

THE CHALLENGE

Limit concurrent operations to two using a semaphore. Track active operations and assert that the limit is never exceeded.

ACCEPTANCE CHECKS

[ ] Every job returns a result

[ ] Maximum active operations <= 2

[ ] No operation remains active after completion

A NUDGE, IF YOU NEED ONE

Use async with semaphore and decrement the active counter in finally.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ASYNC WITHOUT CONFUSION

lesson_14_solution.py

```
import asyncio
async def main():
    limit = asyncio.Semaphore(2)
    active = peak = 0
    async def work(value):
        nonlocal active, peak
        async with limit:
            active += 1
            peak = max(peak, active)
            try:
                await asyncio.sleep(0)
                return value * 2
            finally:
                active -= 1
    result = await asyncio.gather(*(work(i) for i in range(5)))
    assert peak <= 2 and active == 0
    return result
assert asyncio.run(main()) == [0, 2, 4, 6, 8]
```

WHY IT WORKS

A concurrency limit protects a downstream system and your own resources. It is different from a requests-per-second rate limit.

TAKE IT FURTHER

Add cancellation handling and a total timeout, then verify cleanup still runs.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Background jobs

THE GOAL

Separate a quick acknowledgement from durable work.

Long-running work should often be queued rather than keeping an HTTP request open. A background task in the same process can disappear if that process stops. Durable queues need job IDs, states, retries and idempotent processing. A 202 response means accepted, not completed.

lesson_15_example.py

```
jobs = {}
def submit(ident):
    jobs.setdefault(ident, {"state": "queued", "attempts": 0})
    return {"job_id": ident, "state": jobs[ident]["state"]}
assert submit("report-1")["state"] == "queued"
assert submit("report-1")["job_id"] == "report-1"
assert len(jobs) == 1
```

The dictionary models states only. It is not durable and does not coordinate multiple workers. The same job ID does not create a second job in this simplified example.

WATCH FOR THIS

An HTTP success code for job submission must not be displayed as proof that the report or email already completed.

Your turn: Background jobs

THE CHALLENGE

Implement legal transitions queued -> running -> succeeded or failed. Reject a second start after completion.

ACCEPTANCE CHECKS

[ ] Queued can start

[ ] Running can succeed or fail

[ ] Succeeded cannot return to running

A NUDGE, IF YOU NEED ONE

Use a transition map and validate before assignment.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

BACKGROUND JOBS

lesson_15_solution.py

```
ALLOWED = {"queued": {"running"},
           "running": {"succeeded", "failed"},
           "succeeded": set(), "failed": set()}
def transition(job, state):
    if state not in ALLOWED[job["state"]]:
        raise ValueError("Invalid transition")
    return {**job, "state": state}
job = transition({"state": "queued"}, "running")
job = transition(job, "succeeded")
assert job["state"] == "succeeded"
try:
    transition(job, "running")
except ValueError:
    pass
else:
    raise AssertionError("Transition accepted")
```

WHY IT WORKS

A state machine makes retry and recovery decisions visible. A failed job may be retried through a separate explicitly defined transition or a new attempt record.

TAKE IT FURTHER

Design a status endpoint and define how a worker recovers a job abandoned in running state.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Caching and invalidation

THE GOAL

Reuse results without confusing old data with current truth.

A cache stores a result for reuse. It needs a key, expiry policy and invalidation strategy. Cached public data can improve performance, while a wrong key can expose another user's information. Keep authoritative state in durable storage; cache entries should be disposable.

lesson_16_example.py

```
def cache_key(user_id, filters):
    return user_id, tuple(sorted(filters.items()))
assert cache_key(1, {"done": False}) != cache_key(2, {"done": False})
assert cache_key(1, {"a": 1, "b": 2}) == cache_key(1, {"b": 2, "a": 1})
```

The key includes the user scope and canonicalised filters. This example assumes all values are hashable and the chosen filters fully describe the response.

WATCH FOR THIS

Caching an authorisation decision for too long can preserve access after permissions change.

Your turn: Caching and invalidation

THE CHALLENGE

Create a small time-to-live cache with an injected clock value. Expired entries should behave as misses.

ACCEPTANCE CHECKS

[ ] Before expiry -> cached value

[ ] At expiry -> miss

[ ] Unknown key -> miss

A NUDGE, IF YOU NEED ONE

Store value and expires_at together and compare now < expires_at.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CACHING AND INVALIDATION

lesson_16_solution.py

```
class Cache:
    def __init__(self):
        self.data = {}
    def put(self, key, value, now, ttl):
        self.data[key] = (value, now + ttl)
    def get(self, key, now):
        value, expiry = self.data.get(key, (None, 0))
        return value if now < expiry else None
c = Cache()
c.put("public", "report", now=10, ttl=5)
assert c.get("public", 14) == "report"
assert c.get("public", 15) is None
assert c.get("missing", 14) is None
```

WHY IT WORKS

The test avoids real sleeping by supplying time. A production cache also needs eviction, concurrency handling and a decision about whether None can be a stored value.

TAKE IT FURTHER

Add explicit invalidation after a successful task update and test that stale data is no longer returned.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Logs and error handling

THE GOAL

Record enough context to investigate without exposing secrets.

Logs should connect events to a request or job and distinguish expected failures from bugs. Use structured fields where possible. Avoid recording passwords, tokens or full private payloads. Return a safe public message while retaining appropriate internal diagnostics.

lesson_17_example.py

```
import json
record = {"event": "task_created", "request_id": "req-1",
          "task_id": 7, "status": 201}
line = json.dumps(record, sort_keys=True)
assert json.loads(line)["request_id"] == "req-1"
assert "password" not in line
```

The record describes the operation without storing the submitted body. A request identifier lets several log entries be connected.

WATCH FOR THIS

A redaction helper is not a guarantee that every possible secret has been removed.

Your turn: Logs and error handling

THE CHALLENGE

Redact known sensitive top-level fields before serialising a diagnostic record. Do not mutate the original dictionary.

ACCEPTANCE CHECKS

[ ] token and password values become [redacted]

[ ] Other fields remain

[ ] Original values are unchanged

A NUDGE, IF YOU NEED ONE

Build a new dictionary with a small explicit sensitive-key set.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

LOGS AND ERROR HANDLING

lesson_17_solution.py

```
def redact(record):
    sensitive = {"token", "password", "authorization"}
    return {key: "[redacted]" if key.casefold() in sensitive else value
            for key, value in record.items()}
source = {"token": "secret", "event": "login"}
clean = redact(source)
assert clean == {"token": "[redacted]", "event": "login"}
assert source["token"] == "secret"
```

WHY IT WORKS

This exercise handles only top-level known fields. Real logging policies should avoid collecting secrets in the first place and account for nested data and free text.

TAKE IT FURTHER

Add a request ID middleware and confirm that responses and logs share the same generated identifier.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Configuration and releases

THE GOAL

Separate code from deployment-specific values.

Configuration includes database locations, allowed origins and secrets. Read required values at startup and fail clearly if they are missing. A release needs repeatable dependencies, migrations, health checks and a rollback plan. Backups are only useful when restoration has been tested.

lesson_18_example.py

```
def settings(env):
    database = env.get("DATABASE_URL")
    if not database:
        raise ValueError("DATABASE_URL is required")
    return {"database_url": database,
            "debug": env.get("DEBUG", "false").lower() == "true"}
assert settings({"DATABASE_URL": "sqlite:///local.db"})["debug"] is False
```

Passing a mapping into the settings function makes startup rules testable without changing the real environment.

WATCH FOR THIS

Deploying with DEBUG=true can expose internals even when the application otherwise works.

Your turn: Configuration and releases

THE CHALLENGE

Validate a comma-separated allowed-origin configuration. Reject wildcard origins for this credentialed application policy.

ACCEPTANCE CHECKS

[ ] Two origins -> two trimmed values

[ ] Wildcard -> ValueError

[ ] Empty configuration -> ValueError

A NUDGE, IF YOU NEED ONE

Parse, trim, then validate; do not silently substitute a permissive default.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CONFIGURATION AND RELEASES

lesson_18_solution.py

```
def allowed_origins(raw):
    origins = [part.strip() for part in raw.split(",") if part.strip()]
    if not origins or "*" in origins:
        raise ValueError("Explicit origins required")
    return origins
assert allowed_origins("https://a.test, https://b.test") == [
    "https://a.test", "https://b.test"]
for raw in ["", "*"]:
    try:
        allowed_origins(raw)
    except ValueError:
        pass
    else:
        raise AssertionError("Invalid origin policy")
```

WHY IT WORKS

CORS controls which browser origins can read a response; it is not authentication and does not stop direct non-browser requests.

TAKE IT FURTHER

Write a release runbook with migration, smoke test, rollback and backup-restoration steps.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Tests at different boundaries

THE GOAL

Match each test to a specific risk.

Unit tests check domain rules. Integration tests check databases and framework wiring. End-to-end tests check a real user or client journey. Use temporary data and isolated application instances. A test suite is useful when failures tell you which promise broke.

lesson_19_example.py

```
def total(amounts):
    if any(value < 0 for value in amounts):
        raise ValueError("Negative amount")
    return sum(amounts)
assert total([]) == 0
assert total([2, 3]) == 5
try:
    total([-1])
except ValueError:
    pass
else:
    raise AssertionError("Negative amount accepted")
```

These checks use known expected results. A test that computes the expected value with the exact same implementation can repeat the same mistake.

WATCH FOR THIS

Passing in-process tests does not prove that the public hostname, TLS or cloud database is configured correctly.

Your turn: Tests at different boundaries

THE CHALLENGE

Write a FastAPI integration test for valid, invalid and missing input on a simple calculation endpoint.

ACCEPTANCE CHECKS

[ ] Valid input -> expected result

[ ] Wrong type -> 422

[ ] Missing parameter -> 422

A NUDGE, IF YOU NEED ONE

TestClient lets you make requests without a live server.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TESTS AT DIFFERENT BOUNDARIES

lesson_19_solution.py

```
from fastapi import FastAPI
from fastapi.testclient import TestClient
app = FastAPI()
@app.get("/total")
def total(a: int, b: int):
    return {"total": a + b}
c = TestClient(app)
assert c.get("/total?a=2&b=3").json() == {"total": 5}
assert c.get("/total?a=x&b=3").status_code == 422
assert c.get("/total?a=2").status_code == 422
```

WHY IT WORKS

The framework boundary is now tested as a client sees it. Add a real-server smoke test during deployment to catch routing, proxy or configuration problems.

TAKE IT FURTHER

Create an acceptance matrix for the task API: anonymous, owner, other user, invalid payload and missing record.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Project: a task API

THE GOAL

Build a small service and state its limits honestly.

The project creates tasks and retrieves them by ID. This local version uses an in-memory store so you can focus on validation and responses. It is not durable, authenticated or safe for multi-process deployment. The extension path replaces storage and adds ownership checks before public release.

lesson_20_example.py

```
from fastapi import FastAPI
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field, ConfigDict
class TaskInput(BaseModel):
    model_config = ConfigDict(str_strip_whitespace=True)
    title: str = Field(min_length=1, max_length=80)
app = FastAPI()
tasks = {}
@app.post("/tasks", status_code=201)
def create(body: TaskInput):
    ident = len(tasks) + 1
    tasks[ident] = {"id": ident, "title": body.title}
    return tasks[ident]
c = TestClient(app)
assert c.post("/tasks", json={"title": " Read "}).json() == {
    "id": 1, "title": "Read"}
assert c.post("/tasks", json={"title": " "}).status_code == 422
```

The example handles creation only. The lab adds retrieval and checks the full create-then-read path in a standalone application.

WATCH FOR THIS

Do not call a demo production-ready merely because its happy path works. Authentication, ownership, persistence and operational checks remain separate deliverables.

Your turn: Project: a task API

THE CHALLENGE

Build creation and retrieval together. Verify a missing task gives 404, valid creation gives 201 and a read returns the created title.

ACCEPTANCE CHECKS

[ ] POST valid title -> 201

[ ] GET returned ID -> same title

[ ] GET unknown ID -> 404

[ ] Blank title -> 422

A NUDGE, IF YOU NEED ONE

Use HTTPException for missing records and a Pydantic request model for validation.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PROJECT: A TASK API

lesson_20_solution.py

```
from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field, ConfigDict
class TaskInput(BaseModel):
    model_config = ConfigDict(str_strip_whitespace=True)
    title: str = Field(min_length=1, max_length=80)
app, tasks = FastAPI(), {}
@app.post("/tasks", status_code=201)
def create(body: TaskInput):
    ident = len(tasks) + 1
    tasks[ident] = {"id": ident, "title": body.title}
    return tasks[ident]
@app.get("/tasks/{ident}")
def read(ident: int):
    if ident not in tasks:
        raise HTTPException(404, "Task not found")
    return tasks[ident]
c = TestClient(app)
r = c.post("/tasks", json={"title": "Learn"})
assert r.status_code == 201
assert c.get("/tasks/1").json()["title"] == "Learn"
assert c.get("/tasks/99").status_code == 404
assert c.post("/tasks", json={"title": " "}).status_code == 422
```

WHY IT WORKS

The create/read contract is verified. len(tasks)+1 works only for this no-delete, single-process demonstration. Use database-generated identifiers in a persistent service.

TAKE IT FURTHER

Finish a persistent version with SQLite, owner checks, pagination, response models and integration tests. Write down every remaining limitation in its README.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sources & next steps

Use these official references when you need the full API contract or a deeper explanation. Documentation can evolve; check the version used by your own environment.

Flask API

https://flask.palletsprojects.com/en/stable/api/

FastAPI tutorial

https://fastapi.tiangolo.com/tutorial/

FastAPI security

https://fastapi.tiangolo.com/tutorial/security/

FastAPI deployment

https://fastapi.tiangolo.com/deployment/

SQLite transactions

https://docs.python.org/3/library/sqlite3.html

HTTP semantics

https://www.rfc-editor.org/rfc/rfc9110.html

WHAT THE CHECKS ESTABLISH

All 40 companion scripts were executed independently during preparation. Their assertions check the small fixtures shown here. They do not establish production readiness, security under attack, browser accessibility or performance at scale.

YOUR NEXT PROJECT

Choose a small problem you can explain. Write its inputs, outputs and success checks. Build a minimal version, test one failure path, and ask another learner to follow your README. Record what you would improve next.

Keep your notes. Keep experimenting.