CODING INDIA

FastAPI

The Complete Resource Book

Beginner to Advanced, with a focus on AI integration

A hands-on guide to building fast, typed, production-grade Python APIs and AI services.

Published under the Coding India

Dedication

To my parents.

You were never given the gift of education yourselves. My father drove for a living, hands on the wheel through long days, so that his son could sit with books instead.

You could not read the lessons, yet you made sure I could. You turned a driver's seat into a software engineer's chair. Every line of code in this book stands on the roads you drove.

Thank you, Maa and Papa. This one is yours.

About This Book

This is a working reference, not a lecture. Read a chapter, type the code, break it, fix it. The book moves from your first route to authentication, databases, testing, deployment, and a full section on serving AI and large language models over FastAPI. It closes with 50 interview questions and answers so you can prove what you learned.

Everything targets modern FastAPI with Pydantic v2 and Python 3.10+ syntax. Where the older style still appears in real codebases, the book points it out so you recognize it.

How to use it

Beginners: read Parts I to III in order. Intermediate readers: skim to Part IV and V. If you are here for AI work: still skim the foundations, then live in Part VI. Interview prep: the final section, plus the command reference.

Table of Contents

Part I — Foundations

This part takes you from zero to a working, validated API. Read it in order. Every later chapter assumes you can create a route, accept input, and return a typed response.

Chapter 1: What FastAPI Is and Why It Wins

FastAPI is a Python web framework for building APIs. It sits on two libraries: **Starlette** (the web/ASGI layer that handles routing, requests, and responses) and **Pydantic** (data validation using Python type hints). You write ordinary functions with type annotations, and FastAPI turns those annotations into validation, serialization, and automatic documentation.

The three things that make it worth learning

- **Speed of development.** Type hints do double duty: they document your code and validate incoming data. You write less glue code.
- **Async-native.** It runs on ASGI, so it handles thousands of concurrent connections without blocking, which matters a lot when your endpoints call slow things like databases or AI model APIs.
- **Free interactive docs.** Every app ships a Swagger UI at /docs and ReDoc at /redoc, generated from your code with zero extra work.

WSGI vs ASGI (why async matters for AI)

Older frameworks like Flask and Django (classic mode) use WSGI, a synchronous standard: one request occupies a worker until it finishes. If that request waits 8 seconds for an LLM to respond, the worker is frozen for 8 seconds. ASGI, which FastAPI uses, lets a single worker start many slow requests and switch between them while they wait. For AI backends that spend most of their time waiting on model inference or external APIs, this is the difference between serving 5 users and 500 on the same hardware.

Feature	Flask	Django	FastAPI
Standard	WSGI (sync)	WSGI/ASGI	ASGI (async)
Validation	Manual / extensions	Forms/DRF	Built-in (Pydantic)
Auto docs	No	With DRF add-ons	Yes, built-in
Type-hint driven	No	Partial	Yes, core design
Best for AI APIs	Okay	Heavy	Excellent

> **Mental model:** FastAPI = Starlette (the plumbing) + Pydantic (the bouncer that checks data at the door) + a layer that reads your type hints and wires everything together.

Chapter 2: Installation and Project Setup

Use a virtual environment always. It keeps each project's dependencies isolated so one project's package upgrade never breaks another.

```
# 1. Create and activate a virtual environment
python -m venv venv
# macOS / Linux:
source venv/bin/activate
# Windows:
venv\Scripts\activate

# 2. Install FastAPI with the standard extras (includes uvicorn)
pip install "fastapi[standard]"

# Minimal install (you provide the server yourself):
pip install fastapi uvicorn[standard]
```

The **fastapi[standard]** bundle pulls in Uvicorn (the ASGI server that runs your app), the python-multipart package for forms and file uploads, email-validator, and the fastapi CLI. Uvicorn is what actually listens on a port; FastAPI is just the application it runs.

Running the development server

```
# Modern CLI (comes with fastapi[standard]):
fastapi dev main.py       # auto-reload, for development
fastapi run main.py       # production-style, no reload

# The classic way, still everywhere in tutorials and Docker files:
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

In **main:app**, 'main' is the file main.py and 'app' is the FastAPI() object inside it. The --reload flag restarts the server when you save a file; never use it in production because it wastes memory and watches the filesystem.

A sane starting project structure

```
myapi/
  app/
    __init__.py
    main.py          # creates the FastAPI() app, includes routers
    core/
      config.py      # settings (env vars)
      security.py    # auth helpers
    api/
      routes_items.py  # an APIRouter
      routes_users.py
    models.py        # database (SQLAlchemy) models
    schemas.py       # Pydantic request/response models
    db.py            # engine + session
  requirements.txt
  .env
```

Chapter 3: Your First Application

```
from fastapi import FastAPI

app = FastAPI(
    title="Coding India API",
    description="My first FastAPI service",
    version="1.0.0",
)

@app.get("/")
def read_root():
    return {"message": "Hello, Coding India"}
```

Return a Python dict and FastAPI serializes it to JSON automatically, setting the correct content-type header. The **@app.get("/")** decorator is a **path operation**: it binds the HTTP method GET and the URL path / to the function below it.

The HTTP method decorators

- **@app.get** — read data (safe, no side effects).
- **@app.post** — create a new resource.
- **@app.put** — replace a resource entirely.
- **@app.patch** — update part of a resource.
- **@app.delete** — remove a resource.
- Also available: head, options, and trace for completeness.

> Open http://127.0.0.1:8000/docs after starting the server. The Swagger UI you see was generated entirely from the code above. Try the endpoint directly in the browser — no Postman needed.

Chapter 4: Path Parameters

A path parameter is a variable slot inside the URL. Declare it in the path string with braces and as a function argument. The type hint tells FastAPI how to convert and validate it.

```
@app.get("/items/{item_id}")
def get_item(item_id: int):
    return {"item_id": item_id}

# GET /items/42   -> {"item_id": 42}   (int, not string)
# GET /items/abc  -> 422 error, automatically, because abc is not an int
```

Because you annotated item_id as int, FastAPI parses '42' into the integer 42 and returns a clear 422 validation error for anything that is not an integer. You never write that check yourself.

Order matters

Routes are matched top to bottom. A fixed path must be declared before a matching variable path, or the variable one will swallow it.

```
@app.get("/users/me")       # must come first
def read_me():
    return {"user": "current"}

@app.get("/users/{user_id}") # otherwise 'me' is read as a user_id
def read_user(user_id: str):
    return {"user": user_id}
```

Restricting values with Enum

```
from enum import Enum

class ModelName(str, Enum):
    gpt = "gpt"
    claude = "claude"
```

```
    llama = "llama"

@app.get("/models/{name}")
def get_model(name: ModelName):
    return {"model": name.value}
# Only gpt, claude, or llama are accepted; docs show a dropdown.
```

Path validation with Path()

```
from fastapi import Path
from typing import Annotated

@app.get("/items/{item_id}")
def get_item(
    item_id: Annotated[int, Path(ge=1, le=1000, title="Item ID")]
):
    return {"item_id": item_id}
# ge = greater-or-equal, le = less-or-equal. Rejects 0 or 1001.
```

Chapter 5: Query Parameters

Any function argument that is *not* part of the path becomes a query parameter — the part of a URL after the question mark. Give it a default value to make it optional.

```
@app.get("/search")
def search(q: str, limit: int = 10, offset: int = 0):
    return {"q": q, "limit": limit, "offset": offset}

# GET /search?q=fastapi            -> limit=10, offset=0
# GET /search?q=fastapi&limit=5    -> limit=5
# GET /search  (no q)              -> 422, q is required
```

q has no default, so it is **required**. limit and offset have defaults, so they are optional. To make a parameter truly optional and allow it to be missing, type it as Optional and default it to None.

```
from typing import Optional

@app.get("/products")
def products(category: Optional[str] = None):
    if category is None:
        return {"all": True}
    return {"category": category}
```

Rich validation with Query()

```
from fastapi import Query
from typing import Annotated

@app.get("/search")
def search(
    q: Annotated[str, Query(min_length=3, max_length=50,
                            pattern="^[a-z ]+$")] = "",
    tags: Annotated[list[str], Query()] = [],
):
    return {"q": q, "tags": tags}
# GET /search?q=hello&tags=ai&tags=api  -> tags = ['ai','api']
```

> The **Annotated** pattern — Annotated[type, Query(...)] — is the current recommended style. It keeps the real type separate from the validation metadata, which type checkers and editors understand better than the old default-value style.

Chapter 6: Request Bodies with Pydantic

GET requests carry data in the URL. POST, PUT, and PATCH carry a JSON body. To receive a body, declare a Pydantic model and use it as a type hint. FastAPI reads the JSON, validates it against the model, and hands you a fully typed object.

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    description: str | None = None
    in_stock: bool = True

@app.post("/items")
def create_item(item: Item):
    total = item.price * 1.18   # add 18% GST
    return {"name": item.name, "price_with_tax": total}
```

Send this JSON body and it maps straight onto the model. Missing required fields, wrong types, or malformed JSON all produce a 422 with a precise error pointing at the offending field.

```
{
  "name": "Keyboard",
  "price": 1999.0,
  "description": "Mechanical, brown switches"
}
```

Body plus path plus query at once

FastAPI figures out where each argument comes from by its type. Path parameters match the URL, Pydantic models come from the body, and simple types not in the path become query parameters.

```
@app.put("/items/{item_id}")
def update(item_id: int, item: Item, notify: bool = False):
    # item_id -> path, item -> body, notify -> query
    return {"id": item_id, "data": item, "notify": notify}
```

Chapter 7: Pydantic in Depth (v2)

Pydantic is where most of FastAPI's power lives. Modern FastAPI uses Pydantic v2, which is faster and changed some names from v1. Knowing these details separates people who can use FastAPI from people who actually understand it.

Field() for per-field rules and metadata

```
from pydantic import BaseModel, Field

class Product(BaseModel):
    name: str = Field(min_length=1, max_length=100)
    price: float = Field(gt=0, description="Price in INR")
```

```
    quantity: int = Field(default=0, ge=0)
    sku: str = Field(pattern=r"^[A-Z]{3}-\d{4}$")
```

Field validators (custom logic)

```
from pydantic import BaseModel, field_validator, model_validator

class SignUp(BaseModel):
    email: str
    password: str
    confirm: str

    @field_validator("email")
    @classmethod
    def lower_email(cls, v: str) -> str:
        return v.strip().lower()

    @model_validator(mode="after")
    def passwords_match(self):
        if self.password != self.confirm:
            raise ValueError("passwords do not match")
        return self
```

A **field_validator** runs on one field. A **model_validator** with mode='after' runs once the whole object is built, so it can compare fields against each other. Raising ValueError inside either one produces a clean 422 for the client.

Nested and list models

```
class Address(BaseModel):
    city: str
    pincode: str

class User(BaseModel):
    name: str
    addresses: list[Address] = []      # a list of nested models
    metadata: dict[str, str] = {}
```

EmailStr and other special types

```
from pydantic import BaseModel, EmailStr, HttpUrl, SecretStr

class Account(BaseModel):
    email: EmailStr         # validated email format
    website: HttpUrl | None = None
    token: SecretStr        # hidden in logs and repr
```

> **Pydantic v1 to v2 name changes to memorize:** @validator becomes @field_validator; @root_validator becomes @model_validator; .dict() becomes .model_dump(); .json() becomes .model_dump_json(); class Config becomes model_config = ConfigDict(...); orm_mode=True becomes from_attributes=True.

Chapter 8: Response Models and Status Codes

The response_model argument controls the *output* shape. It is a filter and a contract: FastAPI validates your return value against it and drops any field not declared. This is how you stop a password hash from leaking out of a /users endpoint.

```
class UserIn(BaseModel):
    username: str
    password: str         # comes in

class UserOut(BaseModel):
    username: str         # only this goes out

@app.post("/register", response_model=UserOut)
def register(user: UserIn):
    save_to_db(user)
    return user           # password is stripped from the response
```

Setting status codes

```
from fastapi import status

@app.post("/items", status_code=status.HTTP_201_CREATED)
def create():
    return {"created": True}

# Use the status module instead of raw numbers so code reads clearly:
# 200 OK, 201 Created, 204 No Content, 400 Bad Request,
# 401 Unauthorized, 403 Forbidden, 404 Not Found, 422 Validation.
```

Useful response_model helpers

- **response_model_exclude_none=True** — leave out fields that are None.
- **response_model_exclude={'field'}** — drop specific fields.
- **response_model_include={...}** — keep only these fields.

Chapter 9: Error Handling

Return errors deliberately with HTTPException. The status_code and detail become the HTTP status and the JSON error body.

```
from fastapi import HTTPException

@app.get("/items/{item_id}")
def get_item(item_id: int):
    item = db.get(item_id)
    if item is None:
        raise HTTPException(
            status_code=404,
            detail="Item not found",
            headers={"X-Error": "not-found"},
        )
    return item
```

Custom exception handlers

For your own exception types, register a handler once and raise the exception anywhere. This keeps error formatting consistent across the whole app.

```
from fastapi import Request
from fastapi.responses import JSONResponse

class OutOfCreditsError(Exception):
```

```
    def __init__(self, user_id: str):
        self.user_id = user_id

@app.exception_handler(OutOfCreditsError)
def handle_credits(request: Request, exc: OutOfCreditsError):
    return JSONResponse(
        status_code=402,
        content={"error": "no credits", "user": exc.user_id},
    )

@app.post("/generate")
def generate(user_id: str):
    if credits_left(user_id) <= 0:
        raise OutOfCreditsError(user_id)
    return run_model(user_id)
```

This pattern is heavily used in AI backends: one handler for 'rate limited', one for 'model timeout', one for 'no credits', and every endpoint raises them without repeating formatting code.

Part II — Building Real APIs

Foundations get you a route. This part gets you an application: shared logic through dependencies, code split across files, cross-cutting behavior through middleware, and the tools for forms, files, and concurrency.

Chapter 10: Dependency Injection

Dependency injection is FastAPI's best feature after validation. A dependency is a function whose result FastAPI computes and passes into your endpoint. Use it for anything shared: database sessions, the current user, pagination, config, or a loaded AI model.

```
from fastapi import Depends
from typing import Annotated

def pagination(page: int = 1, size: int = 20):
    return {"skip": (page - 1) * size, "limit": size}

@app.get("/items")
def list_items(pg: Annotated[dict, Depends(pagination)]):
    return {"query": pg}
# The page and size query params are validated and reused everywhere.
```

Dependencies with yield (setup and teardown)

A dependency that uses yield runs code before the endpoint, hands over the yielded value, then runs cleanup afterward even if the endpoint raised. This is the standard way to open and close a database session.

```
def get_db():
    db = SessionLocal()
    try:
        yield db        # given to the endpoint
    finally:
        db.close()      # always runs after the response

@app.get("/users/{uid}")
def read(uid: int, db: Annotated[Session, Depends(get_db)]):
    return db.get(User, uid)
```

Class-based dependencies

```
class CommonParams:
    def __init__(self, q: str | None = None, limit: int = 10):
        self.q = q
        self.limit = limit

@app.get("/search")
def search(p: Annotated[CommonParams, Depends()]):
    return {"q": p.q, "limit": p.limit}
```

Sub-dependencies and route-level dependencies

Dependencies can depend on other dependencies, and FastAPI resolves the whole tree, caching each one per request. If a dependency only performs a check (like verifying an API key) and returns nothing you need, attach it to the route instead of the signature.

```
def verify_key(x_api_key: Annotated[str, Header()]):
    if x_api_key != "secret":
        raise HTTPException(401, "bad key")

@app.get("/admin", dependencies=[Depends(verify_key)])
def admin():
    return {"ok": True}

# Apply to a whole app or router:
# app = FastAPI(dependencies=[Depends(verify_key)])
```

> Dependency results are cached within a single request by default. If two dependencies both need get_db, it runs once. Disable with Depends(fn, use_cache=False) in the rare case you want a fresh call.

Chapter 11: Routers and Project Structure

Putting every route in main.py stops scaling around 300 lines. APIRouter is a mini-app you build in its own file and mount onto the main app. Group by resource: users, items, auth, each in its own module.

```
# app/api/routes_items.py
from fastapi import APIRouter

router = APIRouter(prefix="/items", tags=["items"])

@router.get("/")
def list_items():
    return []

@router.get("/{item_id}")
def get_item(item_id: int):
    return {"id": item_id}
```

```
# app/main.py
from fastapi import FastAPI
from app.api import routes_items, routes_users

app = FastAPI()
app.include_router(routes_items.router)
app.include_router(routes_users.router)

# Add a shared prefix and version at include time:
# app.include_router(routes_items.router, prefix="/v1")
```

The **prefix** is prepended to every path in the router, and **tags** group the routes together in the docs UI. This keeps URLs consistent without repeating strings on every function.

Chapter 12: Middleware and CORS

Middleware wraps every request and response. It runs before your endpoint sees the request and after it produces a response — ideal for logging, timing, adding headers, or catching everything.

```
import time
from fastapi import Request

@app.middleware("http")
async def add_timing(request: Request, call_next):
    start = time.perf_counter()
    response = await call_next(request)
    took = time.perf_counter() - start
    response.headers["X-Process-Time"] = f"{took:.4f}"
    return response
```

CORS: the browser rule you will hit on day one

Browsers block a web page on one origin from calling an API on another origin unless the API says it is allowed. If your React frontend on localhost:3000 calls your API on localhost:8000 and the request fails with a CORS error, this is the fix.

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000",
                   "https://myapp.com"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

> Never ship allow_origins=["*"] together with allow_credentials=True to production. It is insecure and browsers reject the combination anyway. List your real frontend domains.

GZip compression

```
from fastapi.middleware.gzip import GZipMiddleware
app.add_middleware(GZipMiddleware, minimum_size=1000)
```

Chapter 13: Forms and File Uploads

HTML forms and file uploads do not send JSON, so they need different declarations. Install python-multipart first (included in fastapi[standard]).

```
from fastapi import Form, File, UploadFile
from typing import Annotated

@app.post("/login")
def login(
    username: Annotated[str, Form()],
    password: Annotated[str, Form()],
):
    return {"user": username}
```

File uploads

```
@app.post("/upload")
async def upload(file: UploadFile):
    contents = await file.read()      # bytes
```

```
    return {
        "filename": file.filename,
        "type": file.content_type,
        "size": len(contents),
    }

# Multiple files:
@app.post("/uploads")
async def uploads(files: list[UploadFile]):
    return [f.filename for f in files]
```

Prefer **UploadFile** over bytes for anything large. UploadFile streams to a spooled temporary file instead of loading the entire upload into memory, so a 500 MB PDF will not crash your worker. This matters directly in AI apps that ingest documents.

Chapter 14: Static Files and Templates

FastAPI is an API framework, but it can serve static assets and render HTML when you need a small frontend or a landing page.

```
from fastapi.staticfiles import StaticFiles
app.mount("/static", StaticFiles(directory="static"), name="static")
```

```
from fastapi import Request
from fastapi.templating import Jinja2Templates

templates = Jinja2Templates(directory="templates")

@app.get("/page")
def page(request: Request):
    return templates.TemplateResponse(
        "home.html", {"request": request, "name": "Coding India"}
    )
```

Chapter 15: Background Tasks

A BackgroundTask runs *after* the response is sent. Use it for quick, fire-and-forget work: sending a confirmation email, writing a log, or invalidating a cache — things the client should not wait for.

```
from fastapi import BackgroundTasks

def write_log(message: str):
    with open("log.txt", "a") as f:
        f.write(message + "\n")

@app.post("/signup")
def signup(email: str, bg: BackgroundTasks):
    bg.add_task(write_log, f"signup: {email}")
    return {"status": "ok"}   # returns immediately
```

> BackgroundTasks run in the same process. They are fine for short jobs but not for heavy or long-running work like training a model or a 2-minute batch job — that needs a real task queue (Celery, RQ, Dramatiq), covered in the AI part.

Chapter 16: Cookies, Headers, and the Raw Request

```
from fastapi import Cookie, Header, Request
from typing import Annotated

@app.get("/context")
def context(
    request: Request,
    session: Annotated[str | None, Cookie()] = None,
    user_agent: Annotated[str | None, Header()] = None,
):
    return {
        "client_ip": request.client.host,
        "method": request.method,
        "session": session,
        "ua": user_agent,
    }
```

Header names with hyphens (User-Agent) map to Python names with underscores (user_agent) automatically. The Request object is the escape hatch: use it when you need raw access, but prefer typed parameters so validation and docs keep working.

Chapter 17: Async and Concurrency

This is the chapter people skip and then get wrong in production. FastAPI supports both def and async def endpoints, and the choice changes how requests share workers.

- **async def** — use when the function awaits async I/O: an async database driver, httpx, an async LLM client. The event loop serves other requests while this one waits.
- **def** (plain) — FastAPI runs it in a separate thread pool so it never blocks the event loop. Use for synchronous libraries and CPU-light work.

The one rule that prevents most async bugs

Never call a blocking, synchronous function directly inside an async def endpoint. A blocking call (a sync database query, time.sleep, a requests.get, or a heavy CPU loop) freezes the entire event loop and every other request stalls behind it.

```
import asyncio, httpx

# GOOD: async endpoint awaiting async I/O
@app.get("/fetch")
async def fetch():
    async with httpx.AsyncClient() as client:
        r = await client.get("https://api.example.com/data")
    return r.json()

# BAD: blocking requests.get inside async def freezes the loop
# @app.get("/bad")
# async def bad():
#     return requests.get(url).json()   # do NOT do this

# If you must run blocking code from async, push it to a thread:
from fastapi.concurrency import run_in_threadpool
@app.get("/heavy")
async def heavy():
    result = await run_in_threadpool(slow_sync_function)
    return result
```

Decision rule: awaiting something? Write async def. Calling a sync library with no await? Write plain def and let FastAPI thread it. Mixing them wrong is the top cause of a 'fast framework that feels slow'.

Part III — Databases and Configuration

An API without persistence is a toy. This part connects FastAPI to a real database with SQLAlchemy, handles schema changes with Alembic, and manages configuration the right way with environment variables.

Chapter 18: SQL Databases with SQLAlchemy

SQLAlchemy is the standard Python ORM. It maps Python classes to database tables. The pattern below is the sync version; it works with any SQL database and is the easiest to learn.

Engine and session

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "sqlite:///./app.db"
# Postgres: "postgresql://user:pass@localhost/dbname"

engine = create_engine(
    DATABASE_URL,
    connect_args={"check_same_thread": False},  # sqlite only
)
SessionLocal = sessionmaker(bind=engine, autoflush=False)
Base = declarative_base()
```

Models and schemas are different things

A SQLAlchemy model describes a database table. A Pydantic schema describes what enters and leaves the API. Keep them separate: the model may have a password hash and timestamps you never expose.

```
from sqlalchemy import Column, Integer, String, Boolean

class User(Base):                # database model
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

Base.metadata.create_all(bind=engine)   # create tables
```

```
from pydantic import BaseModel, EmailStr

class UserCreate(BaseModel):     # incoming
    email: EmailStr
    password: str

class UserRead(BaseModel):       # outgoing
    id: int
    email: EmailStr
    is_active: bool
    model_config = {"from_attributes": True}  # read from ORM object
```

> **from_attributes** = **True** (v1's orm_mode) lets a Pydantic schema read data straight off a SQLAlchemy object, so you can return the ORM row and let response_model convert it.

CRUD in an endpoint

```
from fastapi import Depends, HTTPException
from sqlalchemy.orm import Session
from typing import Annotated

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

DB = Annotated[Session, Depends(get_db)]

@app.post("/users", response_model=UserRead)
def create_user(payload: UserCreate, db: DB):
    user = User(email=payload.email,
                hashed_password=hash_pw(payload.password))
    db.add(user)
    db.commit()
    db.refresh(user)
    return user

@app.get("/users/{uid}", response_model=UserRead)
def read_user(uid: int, db: DB):
    user = db.get(User, uid)
    if not user:
        raise HTTPException(404, "user not found")
    return user
```

The async version (for high concurrency)

When your app is I/O-bound and you want maximum throughput, use SQLAlchemy's async engine with an async driver like asyncpg for Postgres. The shape is the same; the calls are awaited.

```
from sqlalchemy.ext.asyncio import (create_async_engine,
                                    async_sessionmaker, AsyncSession)

engine = create_async_engine(
    "postgresql+asyncpg://user:pass@localhost/db")
AsyncSessionLocal = async_sessionmaker(engine, expire_on_commit=False)

async def get_db():
    async with AsyncSessionLocal() as session:
        yield session

@app.get("/users/{uid}")
async def read_user(uid: int,
                    db: Annotated[AsyncSession, Depends(get_db)]):
    user = await db.get(User, uid)
    return user
```

> Pick one and be consistent. Async DB access pays off when you have many concurrent slow queries or you already run async LLM calls in the same request. For simple CRUD, the sync version is easier and fine — FastAPI threads plain def endpoints so they do not block.

Chapter 19: Migrations with Alembic

create_all builds tables once but cannot evolve them. When you add a column to a live database, you need migrations. Alembic generates and applies versioned schema changes.

```
pip install alembic
alembic init alembic        # creates config + versions folder

# In alembic/env.py, point target_metadata at your Base.metadata
# and set the DB URL. Then:

alembic revision --autogenerate -m "add users table"
alembic upgrade head        # apply migrations
alembic downgrade -1        # roll back one step
```

Autogenerate compares your models to the current database and writes the difference as a migration script. Always read the generated script before applying — autogenerate misses some changes like column renames and treats them as drop-plus-add, which loses data.

Chapter 20: NoSQL with MongoDB (Motor)

For document data, Motor is the async MongoDB driver. FastAPI's async support fits it naturally.

```
import motor.motor_asyncio

client = motor.motor_asyncio.AsyncIOMotorClient("mongodb://localhost")
db = client.mydb

@app.post("/notes")
async def create_note(note: dict):
    result = await db.notes.insert_one(note)
    return {"id": str(result.inserted_id)}

@app.get("/notes")
async def list_notes():
    docs = await db.notes.find().to_list(100)
    for d in docs:
        d["_id"] = str(d["_id"])   # ObjectId is not JSON-safe
    return docs
```

Chapter 21: Configuration and Secrets

Never hard-code database URLs, API keys, or secrets in your source. Load them from environment variables through pydantic-settings, which validates them the same way it validates request bodies.

```
pip install pydantic-settings

from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    app_name: str = "Coding India API"
    database_url: str
    openai_api_key: str
    debug: bool = False

    model_config = SettingsConfigDict(env_file=".env")

settings = Settings()   # reads env vars and .env automatically
```

```
# .env  (never commit this file; add it to .gitignore)
DATABASE_URL=postgresql://user:pass@localhost/prod
OPENAI_API_KEY=sk-...
DEBUG=false
```

Inject settings as a cached dependency so the object is built once and reused, and so tests can override it.

```
from functools import lru_cache
from fastapi import Depends

@lru_cache
def get_settings() -> Settings:
    return Settings()

@app.get("/config")
def show(cfg: Annotated[Settings, Depends(get_settings)]):
    return {"app": cfg.app_name, "debug": cfg.debug}
```

> This is not optional hygiene. A leaked OPENAI_API_KEY or database password committed to a public repo gets scraped and abused within minutes. Environment variables plus .gitignore is the baseline.

Part IV — Security and Authentication

Auth is where APIs get compromised. This part builds a real login system: password hashing, OAuth2 with JWT tokens, protected routes, and role checks. It also covers API keys, the pattern most AI services use.

Chapter 22: Password Hashing

Never store passwords as plain text. Hash them with a slow, salted algorithm so a database leak does not hand over everyone's password. bcrypt via passlib is the standard.

```
pip install "passlib[bcrypt]"

from passlib.context import CryptContext

pwd = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_pw(plain: str) -> str:
    return pwd.hash(plain)

def verify_pw(plain: str, hashed: str) -> bool:
    return pwd.verify(plain, hashed)
```

> Hashing is one-way on purpose. You never decrypt a password; you hash the incoming attempt and compare hashes. bcrypt is deliberately slow to make brute-force attacks expensive.

Chapter 23: OAuth2 with JWT Tokens

The flow: the user posts a username and password to a token endpoint. If valid, you return a signed JWT (JSON Web Token). The client sends that token in the Authorization header on every later request. Your protected routes decode and verify it.

Creating tokens

```
pip install "python-jose[cryptography]"

from datetime import datetime, timedelta, timezone
from jose import jwt, JWTError

SECRET_KEY = settings.secret_key    # long random string, from env
ALGORITHM = "HS256"
EXPIRE_MINUTES = 30

def create_token(data: dict) -> str:
    payload = data.copy()
    expire = datetime.now(timezone.utc) + timedelta(
        minutes=EXPIRE_MINUTES)
    payload.update({"exp": expire})
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)
```

The token endpoint

```
from fastapi.security import OAuth2PasswordRequestForm
from fastapi import Depends, HTTPException, status
```

```
from typing import Annotated

@app.post("/token")
def login(
    form: Annotated[OAuth2PasswordRequestForm, Depends()],
    db: DB,
):
    user = db.query(User).filter(
        User.email == form.username).first()
    if not user or not verify_pw(form.password,
                                 user.hashed_password):
        raise HTTPException(
            status.HTTP_401_UNAUTHORIZED,
            "incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    token = create_token({"sub": str(user.id)})
    return {"access_token": token, "token_type": "bearer"}
```

OAuth2PasswordRequestForm reads the standard username and password form fields, so the /docs UI gets a working Authorize button for free. The 'sub' claim holds the user id; keep tokens small and never put secrets in them, because a JWT payload is only encoded, not encrypted.

Protecting routes: the current-user dependency

```
from fastapi.security import OAuth2PasswordBearer

oauth2 = OAuth2PasswordBearer(tokenUrl="token")

def get_current_user(
    token: Annotated[str, Depends(oauth2)],
    db: DB,
) -> User:
    creds_error = HTTPException(
        status.HTTP_401_UNAUTHORIZED, "invalid token",
        headers={"WWW-Authenticate": "Bearer"})
    try:
        payload = jwt.decode(token, SECRET_KEY,
                             algorithms=[ALGORITHM])
        uid = payload.get("sub")
        if uid is None:
            raise creds_error
    except JWTError:
        raise creds_error
    user = db.get(User, int(uid))
    if user is None:
        raise creds_error
    return user

CurrentUser = Annotated[User, Depends(get_current_user)]

@app.get("/me", response_model=UserRead)
def me(user: CurrentUser):
    return user   # only reachable with a valid token
```

> Now any endpoint that adds the CurrentUser dependency is protected. This single dependency is the backbone of auth in almost every FastAPI codebase you will see.

Chapter 24: Roles, Scopes, and API Keys

Role-based access

```
def require_admin(user: CurrentUser) -> User:
    if user.role != "admin":
        raise HTTPException(403, "admin only")
    return user

@app.delete("/users/{uid}")
def delete_user(uid: int,
                admin: Annotated[User, Depends(require_admin)],
                db: DB):
    db.delete(db.get(User, uid))
    db.commit()
    return {"deleted": uid}
```

API keys (the AI-service pattern)

Machine clients and AI endpoints usually authenticate with a static API key in a header instead of a login flow. It is simpler and stateless.

```
from fastapi.security import APIKeyHeader

api_key_header = APIKeyHeader(name="X-API-Key")

def check_api_key(key: Annotated[str, Depends(api_key_header)]):
    record = db_lookup_key(key)      # hash-compare in real code
    if not record or not record.active:
        raise HTTPException(401, "invalid API key")
    return record

@app.post("/v1/generate",
          dependencies=[Depends(check_api_key)])
def generate(prompt: str):
    return run_model(prompt)
```

Store hashes of API keys, not the keys themselves, exactly as with passwords. Attach usage and rate-limit counters to each key so you can meter and bill — the AI part builds on this.

Part V — Advanced FastAPI

Real-time connections, streaming, testing, and shipping to production. These are the topics that separate a demo from a service people depend on — and several of them are exactly what AI features need.

Chapter 25: WebSockets

HTTP is request-response: the client asks, the server answers, the connection closes. WebSockets keep a two-way connection open so the server can push data any time. This powers chat, live dashboards, and streaming AI conversations.

```
from fastapi import WebSocket, WebSocketDisconnect

@app.websocket("/ws")
async def ws_endpoint(websocket: WebSocket):
    await websocket.accept()
    try:
        while True:
            msg = await websocket.receive_text()
            await websocket.send_text(f"echo: {msg}")
    except WebSocketDisconnect:
        print("client left")
```

A connection manager for broadcasting

```
class ConnectionManager:
    def __init__(self):
        self.active: list[WebSocket] = []

    async def connect(self, ws: WebSocket):
        await ws.accept()
        self.active.append(ws)

    def disconnect(self, ws: WebSocket):
        self.active.remove(ws)

    async def broadcast(self, message: str):
        for conn in self.active:
            await conn.send_text(message)

manager = ConnectionManager()
```

Chapter 26: Streaming Responses and SSE

When output is generated piece by piece — an LLM producing tokens, a large CSV export, a live log — do not make the client wait for the whole thing. Stream it. StreamingResponse sends a generator's output as it is produced.

```
from fastapi.responses import StreamingResponse
import asyncio

async def token_stream():
    for word in ["FastAPI", "streams", "tokens", "live"]:
        yield word + " "
        await asyncio.sleep(0.2)
```

```
@app.get("/stream")
async def stream():
    return StreamingResponse(token_stream(),
                             media_type="text/plain")
```

Server-Sent Events (SSE)

SSE is a simple one-way streaming protocol browsers understand natively. Each message is a line beginning with 'data:'. It is the common way to stream LLM replies to a web frontend without WebSockets.

```
async def sse_events():
    for i in range(5):
        yield f"data: chunk {i}\n\n"
        await asyncio.sleep(1)

@app.get("/events")
async def events():
    return StreamingResponse(
        sse_events(),
        media_type="text/event-stream",
    )
```

Chapter 27: Testing

FastAPI ships a TestClient built on httpx. Tests run your app in-memory with no server, so they are fast enough to run on every save.

```
pip install pytest httpx

from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_read_root():
    r = client.get("/")
    assert r.status_code == 200
    assert r.json() == {"message": "Hello, Coding India"}

def test_create_item():
    r = client.post("/items", json={"name": "Pen", "price": 10})
    assert r.status_code == 201
    assert r.json()["name"] == "Pen"
```

Overriding dependencies in tests

The dependency system makes tests clean: swap the real database for a test one without touching endpoint code.

```
def override_get_db():
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db
```

For async endpoints, use httpx.AsyncClient with ASGITransport inside an async test (with pytest-asyncio) instead of TestClient, so you are exercising the real async path.

Chapter 28: Lifespan Events

Some resources must be created once at startup and cleaned up at shutdown: a database pool, a Redis client, or a loaded ML model. The modern way is the lifespan context manager. The old on_event('startup') and on_event('shutdown') decorators are deprecated.

```
from contextlib import asynccontextmanager
from fastapi import FastAPI

ml_models = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    # startup: load once, not per request
    ml_models["clf"] = load_model("model.pkl")
    yield
    # shutdown: release resources
    ml_models.clear()

app = FastAPI(lifespan=lifespan)

@app.post("/predict")
def predict(x: list[float]):
    return {"y": ml_models["clf"].predict([x]).tolist()}
```

Loading a model on startup instead of inside the endpoint is the single most important performance rule for AI services. Loading a model can take seconds; doing it per request makes your API unusable.

Chapter 29: Deployment

Development uses one Uvicorn process with reload. Production runs multiple workers behind a process manager, usually inside a container.

Running with workers

```
# Multiple Uvicorn workers (rule of thumb: 2 x CPU cores + 1)
uvicorn app.main:app --host 0.0.0.0 --port 8000 --workers 4

# Or Gunicorn managing Uvicorn workers (common in production):
gunicorn app.main:app \
  -w 4 -k uvicorn.workers.UvicornWorker \
  --bind 0.0.0.0:8000
```

A production Dockerfile

```
FROM python:3.12-slim
WORKDIR /code
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY ./app ./app
EXPOSE 8000
CMD ["uvicorn", "app.main:app", \
     "--host", "0.0.0.0", "--port", "8000", "--workers", "4"]
```

- Put Nginx or a cloud load balancer in front for TLS, compression, and buffering.
- Set workers based on cores; each worker is a full process with its own memory, which matters when each one loads a large model.
- Add a /health endpoint that returns 200 so orchestrators know the container is alive.
- Never run with --reload in production.

Chapter 30: Performance, Caching, and Rate Limiting

Caching expensive results

```
import redis.asyncio as redis
import json

r = redis.from_url("redis://localhost")

@app.get("/report/{key}")
async def report(key: str):
    cached = await r.get(key)
    if cached:
        return json.loads(cached)
    data = expensive_query(key)
    await r.set(key, json.dumps(data), ex=300)  # 5 min TTL
    return data
```

Rate limiting

Protect endpoints — especially costly AI ones — from abuse with a limiter such as slowapi.

```
from slowapi import Limiter
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter

@app.post("/v1/generate")
@limiter.limit("10/minute")
async def generate(request: Request, prompt: str):
    return await run_model(prompt)
```

- Serialize responses with a fast library and keep payloads small.
- Use connection pooling for databases; never open a new connection per request.
- Measure before optimizing: add the timing middleware from Chapter 12 and watch real numbers, not guesses.

Chapter 31: Command and CLI Reference

The commands you will actually type, in one place.

Command	What it does
pip install "fastapi[standard]"	Install FastAPI with server + extras
fastapi dev main.py	Run dev server with auto-reload

Command	What it does
fastapi run main.py	Run in production mode
uvicorn main:app --reload	Classic dev server
uvicorn main:app --workers 4	Run with 4 worker processes
gunicorn -k uvicorn.workers.UvicornWorker	Gunicorn managing Uvicorn
alembic init alembic	Set up migrations
alembic revision --autogenerate -m msg	Generate a migration
alembic upgrade head	Apply all migrations
alembic downgrade -1	Roll back one migration
pytest	Run the test suite
docker build -t myapi .	Build the container image
docker run -p 8000:8000 myapi	Run the container

Import cheat sheet

Import	Used for
from fastapi import FastAPI	The app object
from fastapi import Path, Query, Body	Parameter validation
from fastapi import Header, Cookie, Form, File	Other inputs
from fastapi import Depends	Dependency injection
from fastapi import HTTPException, status	Errors and status codes
from fastapi import BackgroundTasks	Fire-and-forget tasks
from fastapi import Request, WebSocket	Raw request / sockets
from fastapi.responses import StreamingResponse	Streaming output
from fastapi.middleware.cors import CORSMiddleware	CORS
from pydantic import BaseModel, Field	Schemas and validation

Part VI — FastAPI for AI

This is why most people learn FastAPI in 2025 and beyond: it has become the default way to put a model behind an API. Everything from the earlier parts now pays off — async for slow inference, dependencies for loaded models, streaming for token output, and auth plus rate limits for cost control. This part builds real AI services end to end.

Chapter 32: Serving a Machine Learning Model

The core pattern: load the model once at startup, then expose a prediction endpoint. Validate inputs and outputs with Pydantic so bad data never reaches the model and clients get a stable contract.

```
from contextlib import asynccontextmanager
from fastapi import FastAPI
from pydantic import BaseModel, Field
import joblib

models = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    models["iris"] = joblib.load("iris_model.pkl")
    yield
    models.clear()

app = FastAPI(lifespan=lifespan)

class IrisInput(BaseModel):
    sepal_length: float = Field(gt=0)
    sepal_width: float = Field(gt=0)
    petal_length: float = Field(gt=0)
    petal_width: float = Field(gt=0)

class Prediction(BaseModel):
    species: str
    confidence: float

@app.post("/predict", response_model=Prediction)
def predict(x: IrisInput):
    features = [[x.sepal_length, x.sepal_width,
                 x.petal_length, x.petal_width]]
    model = models["iris"]
    label = model.predict(features)[0]
    proba = float(model.predict_proba(features).max())
    return Prediction(species=str(label), confidence=proba)
```

CPU-heavy inference must not block the loop

Model inference is CPU work, not I/O. If you run a heavy sync predict inside an async def, it freezes the event loop. Either keep the endpoint as plain def (FastAPI threads it) or push the work to a thread pool from an async endpoint.

```
from fastapi.concurrency import run_in_threadpool

@app.post("/predict-async")
async def predict_async(x: IrisInput):
    result = await run_in_threadpool(heavy_predict, x)
    return result
```

Chapter 33: Calling an LLM API

Most AI features today are a thin FastAPI layer in front of a large language model provider. Your API adds auth, validation, prompt construction, rate limiting, and logging around the model call. Use an async client so one slow model call does not block other requests.

```
from openai import AsyncOpenAI
from pydantic import BaseModel

client = AsyncOpenAI(api_key=settings.openai_api_key)

class ChatRequest(BaseModel):
    message: str
    system: str = "You are a helpful assistant."

@app.post("/chat")
async def chat(req: ChatRequest):
    resp = await client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": req.system},
            {"role": "user", "content": req.message},
        ],
    )
    return {"reply": resp.choices[0].message.content}
```

The same shape with Anthropic's SDK

```
from anthropic import AsyncAnthropic

client = AsyncAnthropic(api_key=settings.anthropic_api_key)

@app.post("/chat-claude")
async def chat_claude(req: ChatRequest):
    msg = await client.messages.create(
        model="claude-sonnet-4-5",
        max_tokens=1024,
        system=req.system,
        messages=[{"role": "user", "content": req.message}],
    )
    return {"reply": msg.content[0].text}
```

> Keep provider keys in environment variables (Chapter 21) and never expose them to the browser. The frontend calls your FastAPI endpoint; your endpoint calls the model. The key stays server-side.

Chapter 34: Streaming LLM Responses

Waiting ten seconds for a full answer feels broken. Streaming tokens as they are generated makes the app feel instant. Combine the LLM's stream with StreamingResponse from Chapter 26.

```
from fastapi.responses import StreamingResponse

@app.post("/chat/stream")
async def chat_stream(req: ChatRequest):
    async def generate():
        stream = await client.chat.completions.create(
            model="gpt-4o-mini",
```

```
            messages=[{"role": "user", "content": req.message}],
            stream=True,
        )
        async for chunk in stream:
            delta = chunk.choices[0].delta.content
            if delta:
                yield delta

    return StreamingResponse(generate(),
                             media_type="text/plain")
```

As Server-Sent Events for a browser frontend

```
import json

@app.post("/chat/sse")
async def chat_sse(req: ChatRequest):
    async def events():
        stream = await client.chat.completions.create(
            model="gpt-4o-mini", stream=True,
            messages=[{"role": "user", "content": req.message}])
        async for chunk in stream:
            token = chunk.choices[0].delta.content or ""
            yield f"data: {json.dumps({'token': token})}\n\n"
        yield "data: [DONE]\n\n"
    return StreamingResponse(events(),
                             media_type="text/event-stream")
```

Chapter 35: Long Jobs with a Task Queue

Fine-tuning, batch embedding, video processing, or any job over a few seconds should not run inside the request. The client would time out and a restart would lose the work. Offload to a task queue (Celery, RQ, or Dramatiq), return a job id immediately, and let the client poll for status.

```
from celery import Celery

celery = Celery("tasks", broker="redis://localhost:6379/0",
                backend="redis://localhost:6379/1")

@celery.task
def run_batch_embeddings(doc_ids: list[int]):
    # heavy work runs in a separate worker process
    return embed_all(doc_ids)

@app.post("/jobs/embed")
def start_job(doc_ids: list[int]):
    task = run_batch_embeddings.delay(doc_ids)
    return {"job_id": task.id, "status": "queued"}

@app.get("/jobs/{job_id}")
def job_status(job_id: str):
    result = celery.AsyncResult(job_id)
    return {"status": result.status,
            "result": result.result if result.ready() else None}
```

> The split is: FastAPI accepts the request and returns a job id fast; a separate Celery worker (its own process, often its own container with a GPU) does the slow work; the client polls the status endpoint or gets notified over a WebSocket.

Chapter 36: A RAG Endpoint

Retrieval-Augmented Generation grounds an LLM in your own documents. The flow: embed the user's question, search a vector database for the most similar chunks, put those chunks into the prompt as context, then ask the model to answer using only that context. FastAPI orchestrates the steps.

```
from pydantic import BaseModel

class Ask(BaseModel):
    question: str
    top_k: int = 4

@app.post("/rag/ask")
async def rag_ask(body: Ask):
    # 1. embed the question
    emb = await client.embeddings.create(
        model="text-embedding-3-small",
        input=body.question)
    vector = emb.data[0].embedding

    # 2. search the vector store for relevant chunks
    chunks = vector_db.search(vector, top_k=body.top_k)
    context = "\n\n".join(c.text for c in chunks)

    # 3. build a grounded prompt
    prompt = (
        "Answer using only the context below.\n\n"
        f"Context:\n{context}\n\n"
        f"Question: {body.question}"
    )

    # 4. generate the answer
    resp = await client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": prompt}])
    return {
        "answer": resp.choices[0].message.content,
        "sources": [c.source for c in chunks],
    }
```

Returning the sources alongside the answer is what makes a RAG endpoint trustworthy: the client can show which documents the answer came from. Popular vector stores that slot into the search step include pgvector (Postgres), Qdrant, Weaviate, Chroma, and Pinecone.

Chapter 37: Document Q&A from an Upload

Combine file uploads (Chapter 13) with the RAG flow: the user uploads a PDF, you extract and chunk the text, embed and store it, then answer questions against it.

```
from fastapi import UploadFile
from pypdf import PdfReader
import io

@app.post("/documents")
async def ingest(file: UploadFile):
    raw = await file.read()
    reader = PdfReader(io.BytesIO(raw))
    text = "".join(p.extract_text() or "" for p in reader.pages)

    chunks = chunk_text(text, size=800, overlap=100)
    for ch in chunks:
        emb = await embed(ch)
        vector_db.add(vector=emb, text=ch,
```

```
                source=file.filename)
    return {"file": file.filename, "chunks": len(chunks)}
```

Chunk with overlap so a sentence split across the boundary still appears whole in at least one chunk. Typical sizes are 500 to 1000 characters with 10 to 15 percent overlap; tune to your documents.

Chapter 38: A Streaming Chat over WebSocket

For a full chat experience with conversation history and live tokens, combine WebSockets (Chapter 25) with LLM streaming (Chapter 34).

```
from fastapi import WebSocket, WebSocketDisconnect

@app.websocket("/ws/chat")
async def ws_chat(ws: WebSocket):
    await ws.accept()
    history = []
    try:
        while True:
            user_msg = await ws.receive_text()
            history.append({"role": "user",
                            "content": user_msg})
            stream = await client.chat.completions.create(
                model="gpt-4o-mini", messages=history,
                stream=True)
            reply = ""
            async for chunk in stream:
                tok = chunk.choices[0].delta.content or ""
                reply += tok
                await ws.send_text(tok)
            history.append({"role": "assistant",
                            "content": reply})
    except WebSocketDisconnect:
        pass
```

Chapter 39: Cost Control and Guardrails for AI Endpoints

AI endpoints cost real money per call and can be abused. Treat cost control as a first-class feature, not an afterthought.

- **Authenticate every AI route** with an API key or token (Chapter 24). Anonymous access to a paid model is a bill waiting to happen.
- **Rate limit per key** (Chapter 30) so one client cannot drain your quota or budget.
- **Cap input and output size** with Pydantic Field(max_length=...) and the model's max_tokens, so a giant prompt cannot run up cost.
- **Cache** repeated or deterministic prompts in Redis to skip paying for the same answer twice.
- **Log tokens and latency** per request so you can attribute cost and catch runaway usage early.
- **Set timeouts** on model calls and return a clean error instead of hanging a connection open.

```
class SafePrompt(BaseModel):
    text: str = Field(min_length=1, max_length=4000)

@app.post("/v1/generate", dependencies=[Depends(check_api_key)])
@limiter.limit("20/minute")
```

```
async def generate(request: Request, body: SafePrompt):
    resp = await client.chat.completions.create(
        model="gpt-4o-mini", max_tokens=500,
        messages=[{"role": "user", "content": body.text}])
    usage = resp.usage
    log_cost(request, usage.total_tokens)
    return {"reply": resp.choices[0].message.content}
```

Chapter 40: Putting It Together — An AI Microservice

A production AI service is rarely one process. A common architecture:

1. **FastAPI app** — handles HTTP and WebSocket traffic, auth, validation, rate limiting. Stays lightweight and async.
2. **Redis** — caching, rate-limit counters, and the Celery broker.
3. **Worker processes** (Celery) — run heavy or long jobs like batch embedding or fine-tuning, often on GPU machines.
4. **Vector database** — stores embeddings for retrieval.
5. **SQL database** — users, API keys, usage records, job metadata.
6. **Model providers or self-hosted models** — the actual inference, called by the app for fast requests and by workers for batch jobs.

FastAPI is the front door tying these together. It stays fast because it delegates: quick model calls run async in the request, slow ones go to workers, repeated ones hit the cache, and everything is metered per API key. Every technique in this book maps onto one box in that diagram.

> If you can build this service — authenticated, streaming, rate-limited, with a RAG endpoint and a background worker — you have the exact skill companies hire FastAPI developers for right now.

Part VII — 50 Interview Questions

Answers are kept tight on purpose — the way you would say them in a room. Read the question, answer it out loud, then check yourself. If you cannot answer one, the chapter it came from is your gap.

Fundamentals (1–10)

1. What is FastAPI built on?
Starlette for the web/ASGI layer and Pydantic for data validation. FastAPI adds the layer that reads type hints to wire them together.

2. Why is FastAPI called fast?
It runs on ASGI with async support and uses Pydantic v2's compiled core, so it handles high concurrency and validates data with low overhead. Its performance is comparable to Node and Go for I/O work.

3. WSGI vs ASGI?
WSGI is synchronous: one request holds a worker until done. ASGI is asynchronous: a worker can handle many concurrent requests, switching between them while they wait on I/O. FastAPI uses ASGI.

4. What is Uvicorn?
An ASGI server. FastAPI is the application; Uvicorn is the process that actually listens on a port and runs it.

5. How does FastAPI generate docs automatically?
It builds an OpenAPI schema from your path operations, type hints, and Pydantic models, then serves Swagger UI at /docs and ReDoc at /redoc.

6. What is a path operation?
The pairing of an HTTP method and a path with a handler function, declared by a decorator like @app.get("/items").

7. How do you return a custom status code?
Pass status_code to the decorator, or return a Response object with the code set. Use the fastapi.status constants for readability.

8. Difference between PUT and PATCH?
PUT replaces the whole resource; PATCH updates only the fields you send. In FastAPI, a PATCH handler typically uses model.model_dump(exclude_unset=True) to apply partial updates.

9. Why does route order matter?
Routes match top to bottom. A fixed path like /users/me must be declared before /users/{id} or the variable route captures 'me'.

10. How do you enable auto-reload in development?
Run fastapi dev main.py, or uvicorn main:app --reload. Never use reload in production.

Parameters and Validation (11–18)

11. How does FastAPI decide if an argument is a path, query, or body parameter?

By type and name. If the name is in the path it is a path parameter; a Pydantic model comes from the body; other simple types become query parameters.

12. How do you make a query parameter optional?

Give it a default value. For a nullable optional, type it as str | None with a default of None.

13. What is Annotated used for?

It attaches validation metadata to a type without changing the type itself, for example Annotated[int, Path(ge=1)]. It is the current recommended style over default-value validators.

14. How do you validate a number range on a path parameter?

Use Path with ge, gt, le, lt, for example Annotated[int, Path(ge=1, le=100)].

15. How do you accept a list in a query string?

Type the parameter as list[str] with Query(); the client repeats the key, like ?tag=a&tag=b.

16. What status code does a validation failure return?

422 Unprocessable Entity, with a JSON body pinpointing the field and the reason.

17. How do you restrict a path parameter to a fixed set of values?

Use a str-based Enum as the type; FastAPI validates against it and shows a dropdown in the docs.

18. How do you receive form data instead of JSON?

Use Form() on each field and install python-multipart. Form and JSON bodies cannot be mixed in the same request.

Pydantic (19–25)

19. Difference between a Pydantic model and a SQLAlchemy model?

A Pydantic model validates and serializes API data. A SQLAlchemy model maps to a database table. Keep them separate so internal fields never leak through the API.

20. What does response_model do?

It defines and enforces the output shape, dropping any field not declared. It is how you stop sensitive fields like password hashes from being returned.

21. field_validator vs model_validator?

field_validator runs on a single field. model_validator with mode='after' runs on the whole object, so it can compare fields against each other.

22. Key Pydantic v1 to v2 changes?

.dict() became .model_dump(); @validator became @field_validator; class Config became model_config = ConfigDict(...); orm_mode became from_attributes.

23. What does from_attributes = True enable?

It lets a Pydantic model read data directly off object attributes, such as a SQLAlchemy row, so you can return the ORM object and have it serialized.

24. How do you exclude None fields from a response?

Set response_model_exclude_none=True on the decorator, or call model_dump(exclude_none=True).

25. How do you validate email or URL fields?

Use EmailStr and HttpUrl types from Pydantic. EmailStr needs the email-validator package, included in fastapi[standard].

Dependencies and Async (26–33)

26. What is a dependency in FastAPI?

A callable whose result FastAPI computes and injects into your endpoint via Depends. Used for DB sessions, current user, config, pagination, and shared checks.

27. Why use a dependency with yield?

The code before yield is setup, the yielded value is injected, and code after yield is teardown that always runs, even on error. It is the standard pattern for database sessions.

28. Are dependencies cached?

Yes, within a single request by default, so a dependency used twice runs once. Disable with use_cache=False.

29. How do you protect a route without needing the dependency's return value?

Attach it to the route with dependencies=[Depends(fn)], or to the whole app or router at creation.

30. async def vs def endpoint — how do you choose?

Use async def when the body awaits async I/O. Use plain def for synchronous libraries; FastAPI runs it in a thread pool so it does not block the event loop.

31. What is the classic async mistake?

Calling a blocking function (sync DB query, requests.get, time.sleep, heavy CPU) directly inside an async def, which freezes the whole event loop and stalls every request.

32. How do you run blocking code from an async endpoint safely?

Await run_in_threadpool from fastapi.concurrency, which offloads the blocking call to a thread.

33. Where should you load an ML model?

Once at startup in a lifespan context manager, storing it in a shared object. Loading per request makes the API unusably slow.

Database, Security, Testing (34–42)

34. How do you manage a database session per request?

A get_db dependency that yields a session and closes it in a finally block, injected via Depends into each endpoint.

35. Why Alembic instead of create_all?

create_all builds tables once but cannot evolve a live schema. Alembic writes versioned migrations you can apply and roll back safely.

36. How do you store passwords?

As bcrypt hashes via passlib, never plain text. You verify by hashing the attempt and comparing hashes; you never decrypt.

37. Outline OAuth2 password flow with JWT.

The user posts credentials to a token endpoint; if valid you return a signed JWT; the client sends it as a Bearer token; a dependency decodes and verifies it to identify the user on protected routes.

38. Is a JWT encrypted?

No, it is signed and only base64-encoded. Anyone can read the payload, so never put secrets in it; the signature only guarantees it was not tampered with.

39. How do AI or machine clients usually authenticate?

With a static API key sent in a header, checked by an APIKeyHeader dependency. Store hashes of keys, not the raw keys.

40. How do you test a FastAPI app?

With TestClient (built on httpx), which runs the app in-memory with no server. For async paths, use httpx.AsyncClient with pytest-asyncio.

41. How do you swap the real database for a test one?

Use app.dependency_overrides to replace the get_db dependency with a test version, without changing endpoint code.

42. How do you configure CORS?

Add CORSMiddleware with an explicit allow_origins list. Never combine allow_origins=['*'] with allow_credentials=True.

Advanced and AI (43–50)

43. When do you use WebSockets over HTTP?

When you need a persistent two-way connection so the server can push data any time — chat, live dashboards, streaming AI conversations.

44. How do you stream an LLM's tokens to a client?

Call the model with stream=True, wrap the async generator in StreamingResponse, and yield each token. For browsers, send them as Server-Sent Events with media_type text/event-stream.

45. Why offload long AI jobs to a task queue?

Requests would time out and a restart would lose in-flight work. A queue like Celery runs the job in a separate worker; the API returns a job id immediately and the client polls for status.

46. Explain a RAG endpoint's steps.

Embed the question, search a vector database for similar chunks, insert those chunks into the prompt as context, then ask the model to answer from that context and return the sources.

47. How do you keep an AI endpoint from blocking under load?

Use an async client for model calls, load models once at startup, and push CPU-heavy inference to a thread pool or a worker process.

48. How do you control cost on AI endpoints?

Authenticate and rate-limit per key, cap input length and max_tokens, cache repeated prompts, set timeouts, and log tokens per request to attribute spend.

49. How do you deploy FastAPI in production?

Run multiple workers via Uvicorn or Gunicorn with the Uvicorn worker class, inside a container, behind Nginx or a load balancer for TLS. No --reload, and a /health endpoint for orchestration.

50. Describe a full AI microservice architecture.

A lightweight async FastAPI app for HTTP and WebSocket traffic, Redis for cache and the task broker, Celery workers for heavy jobs, a vector DB for retrieval, a SQL DB for users and usage, and model providers for inference. FastAPI is the front door that delegates to each.

Closing Note

You now have the full path: a first route, validated inputs, dependencies, databases, authentication, testing, deployment, and a complete toolkit for putting AI behind a fast, safe API. The gap between reading this and being hireable is one thing — build. Type every example. Then build the AI microservice from Chapter 40 and put it on the internet.

This book was written to be the resource I wish existed when I started: plain language, real code, no filler. If it helped you, pass it forward. Teach someone else. That is how Coding India was always meant to work.

Coding India Publications

Built for developers who start with nothing but keep going anyway.