FastAPI

Food Expiration Tracker

A Complete 17-Lesson Course

From first route to production deployment

01 Project Setup & First Route	**02** Pydantic Schemas & Validation	**03** Path & Query Parameters
04 SQLAlchemy Models & Sessions	**05** Full CRUD Operations	**06** Alembic Migrations
07 Response Models & Serialisation	**08** Async Routes	**09** Middleware & Lifespan Events
10 Router Organisation	**11** Dependency Injection	**12** Filtering, Pagination & Envelopes
13 Background Tasks	**14** Uniform Error Handling	**15** Authentication with JWT
16 Testing with Pytest	**17** Deployment with Docker	

LESSON 01

FastAPI Lesson 1 — What is FastAPI & Why Use It

In this lesson you'll get a clear picture of what FastAPI is, where it fits in the Python ecosystem, and why it's a great choice for the app we're going to build throughout this course — a food expiration date tracker. By the end you'll have a running FastAPI server and your first working endpoint.

When you want to let different programs talk to each other over the internet — or let a mobile app or website talk to a backend — you need an **API** (Application Programming Interface).

An API is simply a set of URLs (called **endpoints**) that your app exposes. Other programs can call those URLs to read or change data. For example, our food tracker app will eventually expose endpoints like:

- `GET /foods` — return a list of all stored food items
- `POST /foods` — add a new food item
- `DELETE /foods/1` — remove the food item with ID 1

FastAPI is a Python framework that makes building those endpoints fast, safe, and enjoyable.

There are several Python web frameworks (Flask, Django REST Framework, etc.). Here's why FastAPI stands out:

Feature	FastAPI	Flask	Django REST
Speed (performance)	■ Very fast	Medium	Medium
Automatic docs	■ Built-in	■ Manual	Partial
Data validation	■ Built-in	■ Manual	Partial
Modern Python (type hints)	■ First-class	■ Optional	■ Optional
Learning curve	Low–Medium	Low	High

The two biggest practical wins for us are:

1. **Automatic interactive documentation.** FastAPI reads your code and generates a clickable web UI where you can test every endpoint. No extra work required.
2. **Built-in data validation.** When someone sends data to our API (e.g. a new food item), FastAPI automatically checks it's the right shape before your code even runs.

You'll see the words `async` and `await` in FastAPI code. Here's the short version:

Synchronous (normal Python): Each line runs, finishes, then the next line runs. If one line is waiting (e.g. for a database), everything else waits too.

```
# Synchronous
def get_data():
    result = slow_database_query()  # everything waits here
    return result
```

Asynchronous: While one task is waiting, Python can work on something else. FastAPI is built on top of an async engine called **Starlette**, which is why it's so fast under load.

```
# Asynchronous
async def get_data():
    result = await slow_database_query()  # other requests can run while we wait
    return result
```

For this course: You don't need to master async programming upfront. FastAPI supports both styles, and we'll use plain (synchronous) functions for most of the course, introducing `async` when it genuinely helps. The important thing is recognising the syntax when you see it.

Prerequisites

- Python 3.10 or later (check with `python --version`)
- pip

Create a project folder and virtual environment

```
mkdir food_tracker
cd food_tracker
python -m venv venv
```

Activate the environment:

```
# Windows
venv\Scripts\activate
```

```
# macOS / Linux
source venv/bin/activate
```

Install FastAPI and Uvicorn

```
pip install fastapi uvicorn
```

- **fastapi** — the framework itself
- **uvicorn** — the web server that actually runs your app (FastAPI needs one)

Create a file called `main.py` in your `food_tracker` folder:

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"message": "Welcome to the Food Tracker API"}
```

Let's break this down line by line:

Line	What it does
`from fastapi import FastAPI`	Imports the FastAPI class
`app = FastAPI()`	Creates your application — this is the object everything attaches to
`@app.get("/")`	Registers a route: when someone sends a GET request to `/`, run the function below
`def read_root():`	A plain Python function — your endpoint logic lives here
`return {...}`	FastAPI automatically converts the dictionary to JSON

Run the server

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

- `main` — the filename (`main.py`)
- `app` — the variable name inside that file
- `--reload` — restarts the server automatically whenever you save a change (great during development)

You should see output like:

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process
```

Option 1 — Your browser

Open `http://127.0.0.1:8000` in a browser. You should see:

```
{"message": "Welcome to the Food Tracker API"}
```

Option 2 — Interactive Docs (the FastAPI superpower)

Open `http://127.0.0.1:8000/docs`

You'll see a full interactive UI listing every endpoint in your app. You can click an endpoint, hit **Try it out**, and run it directly in the browser. This is generated automatically from your code — no extra work needed, and it stays in sync as you build.

There's also an alternative docs UI at `http://127.0.0.1:8000/redoc` if you prefer a different style.

Right now our app just returns a welcome message. Over the coming lessons, `main.py` will grow into a full food tracker. Here's a preview of where we're heading:

```
food_tracker/
■
■■■ main.py          ← where we are now
■■■ models.py        ← data shapes (food items, categories)
■■■ database.py      ← connection to SQLite
■■■ routers/
■   ■■■ foods.py     ← all the food-related endpoints
■■■ venv/
```

We'll build this structure piece by piece, so nothing will feel sudden.

---

1. Add a second endpoint to `main.py` that returns a short description of what the food tracker app does:

```
@app.get("/about")
def about():
    return {"app": "Food Tracker", "version": "0.1", "description": "..."}
```

Run the server, visit `/docs`, and use the interactive UI to call your new endpoint.

2. Change the return value of `read_root` to include a `status` key with the value `"running"`. Confirm the server reloads automatically (thanks to `--reload`) and the response updates in your browser without restarting anything.

3. Try visiting `http://127.0.0.1:8000/anything-made-up`. Read the error response FastAPI returns. What HTTP status code does it use, and what does the JSON say? (We'll learn how to customise error responses in Lesson 14.)

- FastAPI is a modern Python framework for building APIs quickly and safely.
- It auto-generates interactive docs from your code — no manual documentation needed.
- `uvicorn` is the server that runs your app; `--reload` makes development smoother.
- An endpoint is just a Python function decorated with `@app.get(...)`, `@app.post(...)`, etc.
- FastAPI supports both sync and async functions — we'll use sync by default and introduce async where it matters.

In **Lesson 2** we'll set up the full project structure, configure a proper virtual environment, and discuss how to organise a FastAPI project as it grows — so the codebase stays clean as we add features to the food tracker.

LESSON 02

FastAPI Lesson 2 — Project Setup

In Lesson 1 you got a single-file FastAPI app running. That's fine as a starting point, but a real project needs a proper structure from the beginning — otherwise things get messy fast. In this lesson we'll set up the full folder layout for the food tracker, install every dependency we'll need across the course, and configure the supporting files (`.env`, `.gitignore`, `requirements.txt`) that every professional Python project should have.

It's tempting to keep everything in `main.py` until the project "gets big enough" to split up. In practice, splitting up later is painful — you end up refactoring import paths, moving logic between files, and untangling circular imports. Setting the structure up now costs five minutes and saves hours later.

By the end of this lesson your project will look like this:

```
food_tracker/
■
■■■ venv/                    ← virtual environment — never commit to git
■
■■■ app/
■   ■■■ __init__.py          ← makes app/ a Python package
■   ■■■ main.py              ← FastAPI app entry point
■   ■■■ database.py          ← database connection (Lesson 8)
■   ■■■ models.py            ← SQLAlchemy models (Lesson 9)
■   ■■■ schemas.py           ← Pydantic schemas (Lesson 5)
■   ■■■ routers/
■       ■■■ __init__.py
■       ■■■ foods.py         ← food item endpoints (Lesson 7)
■
■■■ tests/
■   ■■■ __init__.py
■
■■■ .env                     ← secrets & config — NEVER commit to git
■■■ .gitignore
■■■ requirements.txt
```

We won't fill every file today — but creating the structure now means each subsequent lesson drops into the right place without reorganisation.

With your virtual environment **already activated** from Lesson 1, run:

```
# From inside your food_tracker/ root
mkdir -p app/routers tests

# Create the required __init__.py files
# Windows PowerShell
New-Item app/__init__.py -ItemType File
New-Item app/routers/__init__.py -ItemType File
New-Item tests/__init__.py -ItemType File

# Linux / Raspberry Pi
touch app/__init__.py app/routers/__init__.py tests/__init__.py
```

What is `__init__.py`?

An empty `__init__.py` file tells Python that the folder is a **package** — meaning other files can import from it using dot notation like `from app.routers.foods import router`. Without it, Python won't treat the folder as importable.

Move the `main.py` you created in Lesson 1 into the `app/` folder:

```
# Windows
Move-Item main.py app/main.py

# Linux / Raspberry Pi
mv main.py app/main.py
```

Update the `uvicorn` command to match the new location:

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

The format is always `module.path:variable` — dots instead of slashes, no `.py` extension.

Rather than installing packages one lesson at a time, we'll install everything the course needs now. That way you'll never hit a missing-package error mid-lesson.

```
pip install fastapi "uvicorn[standard]" sqlalchemy alembic python-dotenv
```

Package	Purpose	Used from
`fastapi`	The framework — routes, validation, auto docs	Lesson 1
`uvicorn[standard]`	ASGI web server that runs the app	Lesson 1
`sqlalchemy`	Database ORM — models, queries	Lesson 8
`alembic`	Database migration tool	Lesson 9
`python-dotenv`	Loads `.env` files as environment variables	This lesson

> **Note:** `uvicorn[standard]` (with square brackets) installs extra performance libraries alongside uvicorn. Always prefer this form.

`requirements.txt` records every package your project depends on so that anyone (or you, on another machine) can recreate the exact same environment.

```
pip freeze > requirements.txt
```

To install from it on a fresh machine:

```
python -m venv venv
# activate venv, then:
pip install -r requirements.txt
```

Regenerate this file every time you `pip install` something new. It's the Python equivalent of `package.json` — commit it to version control, use it to share the project.

A `.env` file holds configuration values and secrets — things that change between environments (development vs production) or that must never be committed to git (API keys, database passwords).

Create `.env` in the project root:

```
# .env
APP_NAME=Food Tracker API
APP_VERSION=0.1.0
DEBUG=true
DATABASE_URL=sqlite:///./food_tracker.db
```

Loading .env in FastAPI

Update `app/main.py` to load these values at startup:

```
from fastapi import FastAPI
from dotenv import load_dotenv
import os

load_dotenv()  # reads .env and sets environment variables

app = FastAPI(
    title=os.getenv("APP_NAME", "Food Tracker API"),
    version=os.getenv("APP_VERSION", "0.1.0"),
)


@app.get("/")
def read_root():
    return {
        "message": "Welcome to the Food Tracker API",
        "version": os.getenv("APP_VERSION"),
    }
```

Visit `http://127.0.0.1:8000/docs` after restarting — you'll see your app title and version appear in the docs UI, pulled from `.env`.

If you're using git (recommended), create `.gitignore` in the project root to prevent the virtual environment and secrets from being committed:

```
# .gitignore
venv/
.env
*.env
__pycache__/
*.pyc
*.pyo
.pytest_cache/
*.db
*.sqlite3
.mypy_cache/
dist/
build/
```

The two most important entries are `venv/` (hundreds of megabytes, machine-specific) and `.env` (contains secrets). Never commit either.

Here's the full `app/main.py` at the end of this lesson:

```
from fastapi import FastAPI
from dotenv import load_dotenv
```

```
import os

load_dotenv()

app = FastAPI(
    title=os.getenv("APP_NAME", "Food Tracker API"),
    version=os.getenv("APP_VERSION", "0.1.0"),
    description="Track food items and their expiration dates.",
)


@app.get("/")
def read_root():
    return {
        "message": "Welcome to the Food Tracker API",
        "version": os.getenv("APP_VERSION"),
        "status": "running",
    }


@app.get("/about")
def about():
    return {
        "app": os.getenv("APP_NAME"),
        "version": os.getenv("APP_VERSION"),
        "description": "Track food items and their expiration dates.",
    }
```

Run it with:

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

---

1. Add a `SECRET_KEY` entry to your `.env` file (set it to any random string). Read it in `main.py` and include it in the response from `read_root`. Confirm it appears in the browser. Now delete it from the response — you don't want secrets leaking through an API endpoint. This exercise demonstrates both why `.env` is useful and why it must stay out of git.

2. Create a `config.py` file inside `app/` that centralises all `os.getenv()` calls:

```
# app/config.py
import os
from dotenv import load_dotenv

load_dotenv()

APP_NAME = os.getenv("APP_NAME", "Food Tracker API")
APP_VERSION = os.getenv("APP_VERSION", "0.1.0")
DATABASE_URL = os.getenv("DATABASE_URL", "sqlite:///./food_tracker.db")
DEBUG = os.getenv("DEBUG", "false").lower() == "true"
```

Then update `main.py` to import from `config.py` instead of calling `os.getenv()` directly. This pattern — a single config module — keeps settings in one place as the project grows.

3. Run `pip freeze > requirements.txt` and open the file. Notice that it records not just the packages you installed, but also all their sub-dependencies with pinned version numbers. This is what makes the environment exactly reproducible.

- Put application code inside an `app/` package, not directly in the project root — it keeps imports clean and the project scalable.
- Every folder that needs to be importable requires an `__init__.py` file.
- The `uvicorn` command changes to `uvicorn app.main:app --reload` once `main.py` moves into `app/`.
- `.env` holds configuration and secrets; `python-dotenv` loads them at runtime.
- `requirements.txt` is how you share your project's dependencies — regenerate it every time you install something new.
- `.gitignore` must include `venv/` and `.env` — commit neither.

In **Lesson 3** we'll create our first real endpoint — a `GET /foods` route — and explore how FastAPI handles path operations, HTTP methods, and returning structured data.

LESSON 03

FastAPI Lesson 3 — Your First Real Endpoint: GET Routes & Path Operations

---

In this lesson we build the first real food tracker endpoints — ones that return actual food data rather than just a welcome message. Along the way we'll cover the core vocabulary of FastAPI: **path operations**, **path parameters**, and **query parameters**. We'll also take a proper tour of the Swagger UI that FastAPI generates, since you'll be using it as your primary testing tool throughout the course.

We don't have a database yet — that comes in Lesson 8. For now we'll use a hardcoded list of food items. This is a deliberate choice: it keeps the focus on routing and response structure rather than database complexity. The endpoints we build today will have their internals swapped for real database calls later without changing the routes themselves.

In FastAPI, a **path operation** is the combination of:

- A **path** — the URL the endpoint lives at, e.g. `/foods` or `/foods/1`
- An **operation** — the HTTP method used to call it: `GET`, `POST`, `PUT`, `DELETE`, etc.

Each path operation maps to a Python function. FastAPI calls that function when it receives a matching request.

```
@app.get("/foods")        # path = "/foods", operation = GET
def get_foods():
    ...

@app.post("/foods")       # same path, different operation
def create_food():
    ...
```

The same URL can have multiple operations — each handled by a different function.

Method	Purpose	Idempotent?
`GET`	Read data — never modifies anything	Yes

Method	Purpose	Idempotent?
`POST`	Create a new resource	No
`PUT`	Replace an existing resource entirely	Yes
`PATCH`	Update part of an existing resource	No
`DELETE`	Remove a resource	Yes

This lesson focuses on `GET`. We'll cover the others in Lesson 7 when we build the full CRUD interface for food items.

Until we have a database, we'll work with a hardcoded list of food items in memory. Add this to `app/main.py` (or create a temporary `app/data.py` — either works at this stage):

```
# Hardcoded food items — will be replaced by database calls in Lesson 8
FOODS = [
    {"id": 1, "name": "Milk",        "category": "Dairy",   "expiry_date": "2026-06-10"},
    {"id": 2, "name": "Eggs",        "category": "Dairy",   "expiry_date": "2026-06-14"},
    {"id": 3, "name": "Chicken",     "category": "Meat",    "expiry_date": "2026-06-08"},
    {"id": 4, "name": "Bread",       "category": "Bakery",  "expiry_date": "2026-06-09"},
    {"id": 5, "name": "Cheddar",     "category": "Dairy",   "expiry_date": "2026-06-20"},
    {"id": 6, "name": "Orange Juice","category": "Drinks",  "expiry_date": "2026-06-12"},
]
```

Add this to `app/main.py`:

```
@app.get("/foods")
def get_foods():
    return FOODS
```

Run the server and visit `http://127.0.0.1:8000/foods`. FastAPI automatically serialises the list of dictionaries to a JSON array:

```
[
  {"id": 1, "name": "Milk", "category": "Dairy", "expiry_date": "2026-06-10"},
  {"id": 2, "name": "Eggs", "category": "Dairy", "expiry_date": "2026-06-14"},
  ...
]
```

That's all there is to a basic GET endpoint — a decorator and a function that returns data.

A **path parameter** is a variable embedded in the URL itself. To retrieve a single food item by its ID, the URL looks like `/foods/1` — where `1` is the path parameter.

```
@app.get("/foods/{food_id}")
def get_food(food_id: int):
    for food in FOODS:
        if food["id"] == food_id:
            return food
    return {"error": "Food item not found"}
```

Key points:

- `{food_id}` in the path string is the placeholder — curly braces, no spaces
- `food_id: int` in the function signature tells FastAPI to convert the value to an integer automatically
- If someone calls `/foods/abc`, FastAPI rejects it with a `422 Unprocessable Entity` error before your function even runs — because `"abc"` can't be converted to `int`

Try visiting `http://127.0.0.1:8000/foods/1` and `http://127.0.0.1:8000/foods/999`. Notice the different responses.

A **query parameter** appears after a `?` in the URL: `/foods?category=Dairy`. They're ideal for filtering, sorting, and pagination — anything that narrows or shapes a result without identifying a specific resource.

In FastAPI, any function parameter that isn't part of the path is automatically treated as a query parameter:

```
@app.get("/foods")
def get_foods(category: str = None):
    if category:
        return [f for f in FOODS if f["category"].lower() == category.lower()]
    return FOODS
```

Now:

- `GET /foods` — returns all items
- `GET /foods?category=Dairy` — returns only dairy items
- `GET /foods?category=Meat` — returns only meat items

Required vs Optional Query Parameters

```
# Optional — has a default value of None
def get_foods(category: str = None):

# Required — no default; FastAPI returns a 422 if it's missing
def get_foods(category: str):

# Optional with a typed default
def get_foods(limit: int = 10):
```

For most filtering parameters, optional is the right choice — you want the endpoint to work with or without the filter.

Combining Path and Query Parameters

Both can appear in the same function:

```
@app.get("/foods/{food_id}")
def get_food(food_id: int, include_category: bool = False):
    for food in FOODS:
        if food["id"] == food_id:
            if include_category:
                return food
            return {"id": food["id"], "name": food["name"], "expiry_date": food["expiry_date"]}
    return {"error": "Food item not found"}
```

Call it with `/foods/1?include_category=true` or just `/foods/1`.

```
from fastapi import FastAPI
from app import config

app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
)

FOODS = [
    {"id": 1, "name": "Milk",         "category": "Dairy",   "expiry_date": "2026-06-10"},
    {"id": 2, "name": "Eggs",         "category": "Dairy",   "expiry_date": "2026-06-14"},
    {"id": 3, "name": "Chicken",      "category": "Meat",    "expiry_date": "2026-06-08"},
    {"id": 4, "name": "Bread",        "category": "Bakery",  "expiry_date": "2026-06-09"},
    {"id": 5, "name": "Cheddar",      "category": "Dairy",   "expiry_date": "2026-06-20"},
    {"id": 6, "name": "Orange Juice", "category": "Drinks",  "expiry_date": "2026-06-12"},
]


@app.get("/")
def read_root():
    return {"message": "Welcome to the Food Tracker API", "version": config.APP_VERSION}


@app.get("/foods")
def get_foods(category: str = None):
    if category:
        return [f for f in FOODS if f["category"].lower() == category.lower()]
    return FOODS


@app.get("/foods/{food_id}")
def get_food(food_id: int):
    for food in FOODS:
        if food["id"] == food_id:
            return food
    return {"error": "Food item not found"}
```

Visit `http://127.0.0.1:8000/docs`. You'll see every endpoint listed automatically. Here's what to notice:

Endpoint cards — each path operation appears as a coloured card. GET is green, POST is blue, PUT is orange, DELETE is red.

Try it out — click any endpoint card, then hit "Try it out". Fields appear for every path and query parameter. Fill them in and click "Execute" — FastAPI sends the real request and shows you the response, the status code, and the curl command it used.

Schema — below the response you'll see the inferred response schema. FastAPI has worked out the shape of the data from what your function returns. In later lessons, when we add Pydantic models, this schema becomes precise and exact rather than inferred.

Parameter documentation — any path or query parameter appears in the UI with its type. FastAPI has already read `food_id: int` and `category: str = None` from your function signatures.

The /redoc Alternative

`http://127.0.0.1:8000/redoc` presents the same information in a more readable format — better for sharing with API consumers. Swagger UI (`/docs`) is better for testing during development.

---

1. Add a `limit` query parameter to `GET /foods` that caps the number of results returned:

```
@app.get("/foods")
def get_foods(category: str = None, limit: int = 10):
    results = FOODS
    if category:
        results = [f for f in results if f["category"].lower() == category.lower()]
    return results[:limit]
```

Test it via `/docs` — set `limit` to `2` and confirm only two items come back.

2. Add a `GET /foods/categories` endpoint that returns a deduplicated list of all categories present in the data:

```
@app.get("/foods/categories")
def get_categories():
    return list(set(f["category"] for f in FOODS))
```

Important: this endpoint must be defined **above** `GET /foods/{food_id}` in the file. If it comes after, FastAPI matches `/foods/categories` against `{food_id}` and tries to convert `"categories"` to an integer — which fails. Order matters when path parameters could swallow literal segments.

3. Open `/docs`, expand the `GET /foods` endpoint, click "Try it out", enter `Dairy` in the `category` field, and click Execute. Read the curl command that appears in the response — this is exactly what FastAPI sent under the hood, and you can copy it into a terminal to run it directly.

- A **path operation** is the combination of a URL path and an HTTP method — together they uniquely identify an endpoint.
- **Path parameters** are declared in the URL with curly braces (`{food_id}`) and as typed function arguments. FastAPI validates and converts them automatically.
- **Query parameters** are any function arguments that aren't in the path. Give them a default value to make them optional.
- FastAPI generates `/docs` (Swagger UI) and `/redoc` automatically — use `/docs` to test endpoints interactively during development.
- **Order matters** — define literal path segments (like `/foods/categories`) before parameterised ones (like `/foods/{food_id}`) or FastAPI will match the literal against the parameter first.

In **Lesson 4** we'll look at path and query parameters in greater depth — type coercion, validation constraints, and how to communicate clear errors when parameters are invalid.

LESSON 04

FastAPI Lesson 4 — Path & Query Parameters in Depth: Validation, Types & Constraints

In Lesson 3 we declared parameters using simple type annotations. FastAPI did a lot of work for us — type conversion, basic validation — but there's a whole layer of additional control available through two special functions: `Path()` and `Query()`. This lesson covers how to use them to add constraints, documentation strings, and stricter validation to your parameters, and introduces some richer types (Enums, dates) that make the food tracker's API much more robust.

Consider this endpoint from Lesson 3:

```
@app.get("/foods/{food_id}")
def get_food(food_id: int):
    ...
```

FastAPI already validates that `food_id` is an integer. But it doesn't check whether it's a *sensible* integer. `food_id = -5` or `food_id = 0` will pass validation even though they're meaningless IDs. Similarly:

```
@app.get("/foods")
def get_foods(limit: int = 10):
    ...
```

Nothing stops a caller from passing `limit=99999`, which could return a massive payload and hammer the server. We need constraints.

FastAPI provides `Path()` and `Query()` as drop-in replacements for default values. They accept the same default value as their first argument, then let you add validation rules, metadata, and documentation on top.

```
from fastapi import FastAPI, Path, Query
```

Path() — constraining path parameters

```python
@app.get("/foods/{food_id}")
def get_food(
    food_id: int = Path(..., ge=1, title="Food ID", description="The ID of the food item to retrieve")
):
    ...
```

- `...` (ellipsis) means the parameter is **required** with no default
- `ge=1` means "greater than or equal to 1" — rejects 0 and negatives
- `title` and `description` appear in the auto-generated `/docs`

Query() — constraining query parameters

```python
@app.get("/foods")
def get_foods(
    limit: int = Query(default=10, ge=1, le=100, description="Max number of items to return")
):
    ...
```

- `ge=1` — must be at least 1
- `le=100` — must be at most 100
- Any value outside that range returns a `422 Unprocessable Entity` automatically

Constraint	Meaning	Example
`ge`	Greater than or equal to	`ge=1` — value must be ≥ 1
`gt`	Greater than (strictly)	`gt=0` — value must be > 0
`le`	Less than or equal to	`le=100` — value must be ≤ 100
`lt`	Less than (strictly)	`lt=1000` — value must be < 1000

These work on both `int` and `float` parameters.

```python
@app.get("/foods")
def get_foods(
    category: str = Query(default=None, min_length=2, max_length=50, description="Filter by category name")
):
    ...
```

Constraint	Meaning
`min_length`	Minimum number of characters
`max_length`	Maximum number of characters

Constraint	Meaning
`pattern`	Must match a regex pattern

Pattern example

```
# Only allow alphanumeric category names
category: str = Query(default=None, pattern=r"^[a-zA-Z]+$")
```

This rejects `category=Dairy123` or `category=Da!ry` automatically.

FastAPI supports a range of Python types out of the box, converting and validating them from URL strings automatically.

bool

```
@app.get("/foods")
def get_foods(include_expired: bool = Query(default=False)):
    ...
```

FastAPI accepts `true`, `1`, `on`, `yes` as `True` and `false`, `0`, `off`, `no` as `False` — case-insensitively. So `?include_expired=yes` and `?include_expired=True` both work.

float

```
# A hypothetical "max price" filter
max_price: float = Query(default=None, gt=0)
```

date

```
from datetime import date

@app.get("/foods")
def get_foods(expiring_before: date = Query(default=None)):
    ...
```

FastAPI accepts ISO 8601 date strings (`2026-06-15`) and converts them to Python `date` objects automatically. You can then compare them directly:

```
if expiring_before:
    results = [f for f in FOODS if f["expiry_date"] <= str(expiring_before)]
```

When a parameter should only accept a fixed set of values — like a category or a sort order — an `Enum` is the right tool. It's better than a plain `str` because:

- FastAPI validates the value is one of the allowed options
- The allowed values appear explicitly in `/docs`
- Your IDE can autocomplete them

```
from enum import Enum

class CategoryEnum(str, Enum):
    dairy   = "Dairy"
    meat    = "Meat"
    bakery  = "Bakery"
    drinks  = "Drinks"
    produce = "Produce"
    other   = "Other"
```

Note that it inherits from both `str` and `Enum` — the `str` base is important because FastAPI needs to serialise it to JSON.

Use it as a type annotation:

```
@app.get("/foods")
def get_foods(category: CategoryEnum = Query(default=None)):
    if category:
        return [f for f in FOODS if f["category"] == category.value]
    return FOODS
```

Now `?category=InvalidThing` returns a 422 with a clear message listing the valid options.

In Lesson 3 we wrote `category: str = None`. That works but is slightly informal. The more precise Python typing is:

```
from typing import Optional

category: Optional[str] = Query(default=None)
```

`Optional[str]` is equivalent to `Union[str, None]` — it tells both FastAPI and your IDE that the value can be a string or absent. In Python 3.10+ you can also write `str | None`:

```
category: str | None = Query(default=None)
```

All three forms work in FastAPI. The `Optional[str]` or `str | None` style is preferred because it's explicit about the possibility of `None`.

Both `Path()` and `Query()` accept metadata that enriches the `/docs` UI:

```
food_id: int = Path(
    ...,
    ge=1,
    title="Food ID",
    description="The unique identifier of the food item. Must be a positive integer.",
    example=1,
)


category: Optional[str] = Query(
    default=None,
    title="Category filter",
    description="Filter results to this category only. Case-insensitive.",
    example="Dairy",
)
```

`example` sets the value pre-filled in the `/docs` "Try it out" form — a small touch that makes the API much easier to explore.

Here's the updated `get_foods` endpoint using everything from this lesson:

```
from fastapi import FastAPI, Path, Query
from typing import Optional
from enum import Enum
from datetime import date
from app import config

app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
)


class CategoryEnum(str, Enum):
    dairy   = "Dairy"
    meat    = "Meat"
    bakery  = "Bakery"
    drinks  = "Drinks"
    produce = "Produce"
    other   = "Other"


FOODS = [
    {"id": 1, "name": "Milk",         "category": "Dairy",   "expiry_date": "2026-06-10"},
    {"id": 2, "name": "Eggs",         "category": "Dairy",   "expiry_date": "2026-06-14"},
    {"id": 3, "name": "Chicken",      "category": "Meat",    "expiry_date": "2026-06-08"},
    {"id": 4, "name": "Bread",        "category": "Bakery",  "expiry_date": "2026-06-09"},
    {"id": 5, "name": "Cheddar",      "category": "Dairy",   "expiry_date": "2026-06-20"},
    {"id": 6, "name": "Orange Juice", "category": "Drinks",  "expiry_date": "2026-06-12"},
]


@app.get("/foods/categories")
```

```
def get_categories():
    return list(set(f["category"] for f in FOODS))


@app.get("/foods")
def get_foods(
    category: Optional[CategoryEnum] = Query(
        default=None,
        description="Filter by food category.",
        example="Dairy",
    ),
    expiring_before: Optional[date] = Query(
        default=None,
        description="Return only items expiring on or before this date (YYYY-MM-DD).",
        example="2026-06-15",
    ),
    limit: int = Query(
        default=10,
        ge=1,
        le=100,
        description="Maximum number of items to return.",
    ),
):
    results = FOODS

    if category:
        results = [f for f in results if f["category"] == category.value]
    # ... (truncated for readability)
```

Open `/docs` — notice how much richer the parameter descriptions are now, and how the `CategoryEnum` shows a dropdown of valid values in the "Try it out" form.

When validation fails, FastAPI returns a structured `422 Unprocessable Entity` response. For example, calling `GET /foods?limit=-1`:

```
{
  "detail": [
    {
      "type": "greater_than_equal",
      "loc": ["query", "limit"],
      "msg": "Input should be greater than or equal to 1",
      "input": "-1",
      "ctx": {"ge": 1}
    }
  ]
}
```

This tells the caller exactly what went wrong, where (`query → limit`), and what the constraint is (`ge: 1`). Your code didn't have to write a single line of error handling to produce this.

---

1. Add an `expiring_after` query parameter to `get_foods` so callers can specify a date range:

```
expiring_after: Optional[date] = Query(default=None, description="Return items expiring after this date.")
```

Then filter results where `food["expiry_date"] >= str(expiring_after)`. Test in `/docs` with a date range that includes only some items.

2. Add a `sort_by` parameter using an Enum with two valid values — `"name"` and `"expiry_date"`:

```
class SortByEnum(str, Enum):
    name        = "name"
    expiry_date = "expiry_date"
```

Use it to sort the results before applying the limit. Check that `/docs` shows a dropdown rather than a free-text field.

3. Trigger a 422 error deliberately. Call `GET /foods?limit=0` and `GET /foods?category=InvalidCategory` via `/docs`. Read the error response carefully — notice the `loc` field tells you exactly which parameter failed and the `msg` tells you why.

- `Path()` and `Query()` replace simple default values and add constraints, metadata, and documentation in one place.
- Numeric constraints (`ge`, `gt`, `le`, `lt`) and string constraints (`min_length`, `max_length`, `pattern`) are declared inline — no manual validation needed.
- `Enums` lock a parameter to a fixed set of values and automatically appear as dropdowns in `/docs`.
- FastAPI accepts `date` objects directly — it parses ISO date strings from the URL and converts them for you.
- A `422 Unprocessable Entity` response with a structured body is returned automatically when any constraint is violated — no error-handling code needed from you.
- Use `Optional[T]` or `T | None` rather than a bare `= None` for optional parameters — it's more explicit and better documented.

In **Lesson 5** we introduce Pydantic models — the foundation of request body validation and structured response types. This is where FastAPI's data validation goes from "useful" to "remarkable".

LESSON 05

FastAPI Lesson 5 — Request Bodies & Pydantic Models

So far, every endpoint we've built returns data. In this lesson we define the *shape* of that data precisely, using **Pydantic models** — and we start accepting structured data as input too. Pydantic is the validation library FastAPI is built on. Once you understand it, you understand the core of how FastAPI works. By the end of this lesson we'll have a proper `FoodItem` model, a `schemas.py` file, and endpoints that both accept and return well-defined, validated data.

Pydantic is a Python library that defines data shapes using ordinary Python classes and type annotations. When you create an instance of a Pydantic model, it validates the input against the declared types — raising a descriptive error if anything is wrong.

```
from pydantic import BaseModel
from datetime import date

class FoodItem(BaseModel):
    name: str
    expiry_date: date
```

That's all it takes to define a model. Pydantic handles:

- **Type conversion** — `"2026-06-15"` becomes a Python `date` object
- **Validation** — missing required fields raise errors immediately
- **Serialisation** — models convert cleanly to and from JSON

FastAPI uses Pydantic models for two distinct purposes: **request bodies** (data coming *in*) and **response models** (data going *out*). We'll cover both in this lesson.

A **request body** is data sent in the body of an HTTP request — typically as JSON. It's how a caller provides structured data when creating or updating a resource. For example, to add a new food item:

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

```
{
  "name": "Milk",
  "category": "Dairy",
  "expiry_date": "2026-06-10"
}
```

Query parameters are fine for simple filters. Request bodies are for structured data with multiple fields — exactly what adding a food item requires.

Create `app/schemas.py` — this is where all Pydantic models will live:

```
# app/schemas.py
from pydantic import BaseModel, Field
from datetime import date
from typing import Optional
from enum import Enum


class CategoryEnum(str, Enum):
    dairy   = "Dairy"
    meat    = "Meat"
    bakery  = "Bakery"
    drinks  = "Drinks"
    produce = "Produce"
    other   = "Other"


class FoodItemCreate(BaseModel):
    """Schema for creating a new food item — sent in the request body."""
    name:        str
    category:    CategoryEnum
    expiry_date: date
    notes:       Optional[str] = None


class FoodItemResponse(BaseModel):
    """Schema for returning a food item in a response."""
    id:          int
    name:        str
    category:    CategoryEnum
    expiry_date: date
    notes:       Optional[str] = None
```

Why two models?

Notice there are two separate classes — `FoodItemCreate` and `FoodItemResponse`. This is a deliberate pattern:

- `FoodItemCreate` — what the caller sends. No `id` field, because the server assigns that.
- `FoodItemResponse` — what the API returns. Includes `id` because it's been assigned.

Keeping them separate gives you full control over what comes in versus what goes out, without awkward optional hacks.

In Pydantic, a field is **required** if it has no default value, and **optional** if it does.

```
class FoodItemCreate(BaseModel):
    name:        str              # required — must be provided
    category:    CategoryEnum     # required — must be provided
    expiry_date: date             # required — must be provided
    notes:       Optional[str] = None  # optional — defaults to None if omitted
```

If a caller sends a request body without `name`, Pydantic raises a validation error immediately — before your function runs. The error is returned to the caller as a `422 Unprocessable Entity` with a clear message.

Just like `Query()` and `Path()` in Lesson 4, `Field()` adds constraints and documentation to individual model fields:

```
from pydantic import BaseModel, Field
from datetime import date
from typing import Optional


class FoodItemCreate(BaseModel):
    name: str = Field(
        ...,
        min_length=1,
        max_length=100,
        description="The name of the food item.",
        example="Milk",
    )
    category: CategoryEnum = Field(
        ...,
        description="The category this food belongs to.",
        example="Dairy",
    )
    expiry_date: date = Field(
        ...,
        description="The expiration date in ISO format (YYYY-MM-DD).",
        example="2026-06-10",
    )
    notes: Optional[str] = Field(
        default=None,
        max_length=500,
        description="Optional notes about the item.",
        example="Full fat, 2 litres",
    )
```

The `...` (ellipsis) as the first argument means required — the same convention as `Path()`. The `example` values appear pre-filled in the `/docs` "Try it out" form.

To use a Pydantic model as a request body, declare it as a function parameter with a type annotation — no decorator needed. FastAPI detects it automatically:

```
from app.schemas import FoodItemCreate, FoodItemResponse

@app.post("/foods")
def create_food(food: FoodItemCreate):
    # food is a validated FoodItemCreate instance
    print(food.name)         # "Milk"
    print(food.expiry_date)  # date(2026, 6, 10) — already a Python date object
    print(food.category)     # CategoryEnum.dairy
    print(food.model_dump()) # {'name': 'Milk', 'category': <CategoryEnum.dairy: 'Dairy'>, ...}

    # For now, simulate assigning an ID and storing it
    new_food = {"id": len(FOODS) + 1, **food.model_dump()}
    FOODS.append(new_food)
    return new_food
```

FastAPI knows this is a request body (not a query parameter) because `FoodItemCreate` inherits from `BaseModel`. Query parameters are plain types like `str` and `int`; body parameters are `BaseModel` subclasses.

model_dump()

`food.model_dump()` converts the Pydantic model instance to a plain Python dictionary. You'll use this when storing data or passing it to a database. Note: in Pydantic v1 this was called `.dict()` — the food tracker uses Pydantic v2 (installed with modern FastAPI), where it's `.model_dump()`.

A **response model** tells FastAPI what shape the response should have. Declare it in the `@app.get()` or `@app.post()` decorator:

```
@app.get("/foods", response_model=list[FoodItemResponse])
def get_foods():
    return FOODS

@app.get("/foods/{food_id}", response_model=FoodItemResponse)
def get_food(food_id: int):
    for food in FOODS:
        if food["id"] == food_id:
            return food
    ...
```

With `response_model` set, FastAPI:

1. **Validates** the response against the schema before sending it
2. **Filters out** any extra fields not in the model — even if your function returns them
3. **Documents** the exact response shape in `/docs` and `/redoc`

This is the mechanism that prevents accidental data leaks — if you later add a `password_hash` field to your database model but forget to exclude it, the response model silently drops it.

For more complex validation rules, Pydantic provides field validators — functions that run after type conversion:

```
from pydantic import BaseModel, Field, field_validator
from datetime import date


class FoodItemCreate(BaseModel):
    name:        str
    category:    CategoryEnum
    expiry_date: date
    notes:       Optional[str] = None

    @field_validator("expiry_date")
    @classmethod
    def expiry_must_be_future(cls, v: date) -> date:
        if v < date.today():
            raise ValueError("Expiry date must be today or in the future")
        return v

    @field_validator("name")
    @classmethod
    def name_must_not_be_blank(cls, v: str) -> str:
        if not v.strip():
            raise ValueError("Name cannot be blank or whitespace only")
        return v.strip()
```

These validators run automatically when a `FoodItemCreate` is constructed — whether from an API request, a test, or anywhere else in your code. The error messages appear in the `422` response body.

Here is the full `app/schemas.py` at the end of this lesson:

```
# app/schemas.py
from pydantic import BaseModel, Field, field_validator
from datetime import date
from typing import Optional
from enum import Enum


class CategoryEnum(str, Enum):
    dairy   = "Dairy"
    meat    = "Meat"
    bakery  = "Bakery"
    drinks  = "Drinks"
    produce = "Produce"
    other   = "Other"


class FoodItemCreate(BaseModel):
    """Data required to create a new food item."""
```

```
    name: str = Field(
        ..., min_length=1, max_length=100,
        description="Name of the food item.", example="Milk",
    )
    category: CategoryEnum = Field(
        ..., description="Food category.", example="Dairy",
    )
    expiry_date: date = Field(
        ..., description="Expiry date (YYYY-MM-DD).", example="2026-06-10",
    )
    notes: Optional[str] = Field(
        default=None, max_length=500,
        description="Optional notes.", example="Full fat, 2 litres",
    )

    @field_validator("expiry_date")
    @classmethod
    def expiry_must_be_future(cls, v: date) -> date:
        if v < date.today():
            raise ValueError("Expiry date must be today or in the future")
        return v

    @field_validator("name")
    @classmethod
    def name_must_not_be_blank(cls, v: str) -> str:
        if not v.strip():
            raise ValueError("Name cannot be blank or whitespace only")
        return v.strip()


class FoodItemResponse(BaseModel):
    """Data returned when reading a food item."""
    id:          int
    name:        str
    category:    CategoryEnum
    expiry_date: date
    notes:       Optional[str] = None
```

Update `app/main.py` to import from `schemas` and wire the response models:

```
from fastapi import FastAPI, Path, Query
from typing import Optional
from datetime import date
from app import config
from app.schemas import FoodItemCreate, FoodItemResponse, CategoryEnum

app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
)

FOODS = [
    {"id": 1, "name": "Milk",    "category": "Dairy",  "expiry_date": "2026-06-10", "notes": None},
    {"id": 2, "name": "Eggs",    "category": "Dairy",  "expiry_date": "2026-06-14", "notes": None},
    {"id": 3, "name": "Chicken", "category": "Meat",   "expiry_date": "2026-06-08", "notes": None},
    {"id": 4, "name": "Bread",   "category": "Bakery", "expiry_date": "2026-06-09", "notes": None},
```

```
    {"id": 5, "name": "Cheddar",      "category": "Dairy",  "expiry_date": "2026-06-20", "notes": None},
    {"id": 6, "name": "Orange Juice", "category": "Drinks", "expiry_date": "2026-06-12", "notes": None},
]


@app.get("/foods/categories")
def get_categories() -> list[str]:
    return list(set(f["category"] for f in FOODS))


@app.get("/foods", response_model=list[FoodItemResponse])
def get_foods(
    category: Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]  = Query(default=None),
    limit: int = Query(default=10, ge=1, le=100),
):
    results = FOODS
    if category:
        results = [f for f in results if f["category"] == category.value]
    if expiring_before:
        results = [f for f in results if f["expiry_date"] <= str(expiring_before)]
    return results[:limit]


@app.post("/foods", response_model=FoodItemResponse)
def create_food(food: FoodItemCreate):
    new_food = {"id": len(FOODS) + 1, **food.model_dump(mode="json")}
    FOODS.append(new_food)
    return new_food


@app.get("/foods/{food_id}", response_model=FoodItemResponse)
def get_food(
    food_id: int = Path(..., ge=1, example=1),
):
    for food in FOODS:
        if food["id"] == food_id:
            return food
    return {"error": "Food item not found"}
```

Run the server and open `/docs`. You'll notice several improvements:

1. **POST /foods** now has a request body editor with the correct JSON schema pre-populated
2. **GET /foods** and **GET /foods/{food_id}** show an exact response schema under "Responses" — not an inference, but the actual `FoodItemResponse` definition
3. The **Schemas** section at the bottom of `/docs` lists `FoodItemCreate` and `FoodItemResponse` as named, reusable schemas

Try submitting an invalid request body — for example, omitting `name`, or supplying an expiry date in the past. Read the 422 response.

---

1. Add a `quantity` field to `FoodItemCreate` — an optional integer defaulting to `1` with a minimum value of 1. Add it to `FoodItemResponse` too. Use `Field()` with `ge=1` and an appropriate description. Test that submitting `"quantity": 0` returns a 422.

```
quantity: int = Field(default=1, ge=1, description="Number of units.", example=2)
```

2. Add a `@field_validator` that ensures `name` is in title case when stored — so `"MILK"` and `"milk"` are both normalised to `"Milk"`:

```
@field_validator("name")
@classmethod
def normalise_name(cls, v: str) -> str:
    return v.strip().title()
```

Submit `"name": "whole milk"` via `/docs` and confirm the response returns `"name": "Whole Milk"`.

3. What happens when you call `GET /foods/{food_id}` with an ID that doesn't exist? Right now the function returns `{"error": "Food item not found"}` — but that doesn't match `FoodItemResponse` and will cause a validation error on the response. Make a note of this problem: we'll solve it properly in Lesson 14 using `HTTPException`. For now, just observe what error is produced.

- **Pydantic** `BaseModel` defines the shape of data — both incoming request bodies and outgoing responses.
- A function parameter typed as a `BaseModel` subclass is automatically treated as a **request body** by FastAPI.
- `FoodItemCreate` (no `id`) and `FoodItemResponse` (with `id`) are kept separate — precise control over what comes in vs what goes out.
- `Field()` adds validation constraints and documentation to individual model fields, just as `Query()` and `Path()` do for parameters.
- `response_model` in the decorator validates responses, filters extra fields, and documents the response shape in `/docs`.
- `field_validator` functions run custom validation logic after type conversion — business rules like "expiry date must be in the future".
- All schemas belong in `app/schemas.py` — keep them out of `main.py` to avoid clutter.

In **Lesson 6** we'll go deeper into Pydantic — model configuration, computed fields, model inheritance, and how to handle partial updates cleanly using a separate "update" schema.

LESSON 06

FastAPI Lesson 6 — Data Validation in Depth: Model Config, Inheritance & Update Schemas

---

Lesson 5 introduced Pydantic models and got us to a working `FoodItemCreate` and `FoodItemResponse`. This lesson goes deeper into how those models should be structured as the project grows. We'll cover **model inheritance** to eliminate duplication, **update schemas** to handle PATCH requests cleanly, **cross-field validation** for rules that span multiple fields, and **model configuration** — particularly the `from_attributes` setting that makes our models database-ready for Lesson 8.

Looking at `schemas.py` from Lesson 5, both `FoodItemCreate` and `FoodItemResponse` declare the same four fields:

```
class FoodItemCreate(BaseModel):
    name:        str
    category:    CategoryEnum
    expiry_date: date
    notes:       Optional[str] = None

class FoodItemResponse(BaseModel):
    id:          int           # ← only difference
    name:        str           # ← duplicated
    category:    CategoryEnum  # ← duplicated
    expiry_date: date          # ← duplicated
    notes:       Optional[str] = None  # ← duplicated
```

When we add a new field (like `quantity` from the Lesson 5 exercise), we have to add it in two places. With more schemas — and we'll add one this lesson — that becomes three places. The solution is inheritance.

Define a **base model** containing the shared fields, then inherit from it:

```
# app/schemas.py

class FoodItemBase(BaseModel):
    """Fields shared by all FoodItem schemas."""
```

```
    name:        str           = Field(..., min_length=1, max_length=100, example="Milk")
    category:    CategoryEnum  = Field(..., example="Dairy")
    expiry_date: date          = Field(..., example="2026-06-10")
    notes:       Optional[str] = Field(default=None, max_length=500)

    @field_validator("expiry_date")
    @classmethod
    def expiry_must_be_future(cls, v: date) -> date:
        if v < date.today():
            raise ValueError("Expiry date must be today or in the future")
        return v

    @field_validator("name")
    @classmethod
    def name_must_not_be_blank(cls, v: str) -> str:
        if not v.strip():
            raise ValueError("Name cannot be blank or whitespace only")
        return v.strip().title()


class FoodItemCreate(FoodItemBase):
    """Request body for creating a food item. No id — server assigns it."""
    pass  # inherits everything from FoodItemBase


class FoodItemResponse(FoodItemBase):
    """Response body for returning a food item. Includes server-assigned id."""
    id: int
```

Now `FoodItemCreate` and `FoodItemResponse` are both thin — they only declare what makes them different from the base. The validators in `FoodItemBase` are inherited automatically.

Inheritance hierarchy

```
FoodItemBase     ← shared fields + validators
■■■ FoodItemCreate   ← no id (input)
■■■ FoodItemResponse ← adds id (output)
■■■ FoodItemUpdate   ← all fields optional (PATCH) ← we'll add this next
```

A PATCH request updates only the fields the caller provides. The other fields stay unchanged. This creates a problem: our `FoodItemCreate` makes all fields required. Sending a PATCH body with only `{"notes": "semi-skimmed"}` would fail validation because `name`, `category`, and `expiry_date` are all missing.

The clean solution is a dedicated update schema where every field is optional:

```
class FoodItemUpdate(BaseModel):
    """
    Request body for partially updating a food item.
    All fields are optional — only provided fields are updated.
    """
    name:        Optional[str]          = Field(default=None, min_length=1, max_length=100)
    category:    Optional[CategoryEnum] = None
    expiry_date: Optional[date]         = None
    notes:       Optional[str]          = Field(default=None, max_length=500)
```

```
    @field_validator("name")
    @classmethod
    def name_must_not_be_blank(cls, v: Optional[str]) -> Optional[str]:
        if v is not None and not v.strip():
            raise ValueError("Name cannot be blank or whitespace only")
        return v.strip().title() if v else v

    @field_validator("expiry_date")
    @classmethod
    def expiry_must_be_future(cls, v: Optional[date]) -> Optional[date]:
        if v is not None and v < date.today():
            raise ValueError("Expiry date must be today or in the future")
        return v
```

Why not inherit from `FoodItemBase`? Because `FoodItemBase` has required fields. To make all fields optional we'd have to override every one — more work than a fresh class. The validators are duplicated, but they're short, and the clarity of intent ("all fields are optional") is worth it.

Using FoodItemUpdate in a PATCH endpoint

```
@app.patch("/foods/{food_id}", response_model=FoodItemResponse)
def update_food(food_id: int, updates: FoodItemUpdate):
    for food in FOODS:
        if food["id"] == food_id:
            # model_dump(exclude_unset=True) only returns fields the caller provided
            update_data = updates.model_dump(exclude_unset=True, mode="json")
            food.update(update_data)
            return food
    return {"error": "Food item not found"}
```

`exclude_unset=True` is the key detail. It returns only the fields the caller actually sent — not all the optional ones that defaulted to `None`. Without it, a PATCH body of `{"notes": "semi-skimmed"}` would also set `name`, `category`, and `expiry_date` to `None`.

```
# Caller sends: {"notes": "semi-skimmed"}
updates.model_dump()                    # {"name": None, "category": None, "expiry_date": None, "notes": "semi-skimme
updates.model_dump(exclude_unset=True)  # {"notes": "semi-skimmed"}  ← only what was sent
```

A `@field_validator` validates one field in isolation. A `@model_validator` runs after all fields are set and can compare them against each other. This is the right tool for rules like "expiry date must be after the purchase date" — something neither field can enforce on its own.

```
from pydantic import BaseModel, model_validator
from datetime import date
from typing import Optional, Self


class FoodItemCreate(FoodItemBase):
    purchase_date: Optional[date] = None

    @model_validator(mode="after")
    def expiry_after_purchase(self) -> Self:
```

```
        if self.purchase_date and self.expiry_date:
            if self.expiry_date <= self.purchase_date:
                raise ValueError("Expiry date must be after the purchase date")
        return self
```

`mode="after"` means the validator runs after all individual fields have been validated and converted. `Self` is the return type — the validator must return the model instance (`self`). Raising a `ValueError` is returned to the caller as a `422`.

Pydantic v2 uses a `model_config` class attribute to configure how a model behaves. The most important setting for our project is `from_attributes`:

```
from pydantic import BaseModel, ConfigDict

class FoodItemResponse(FoodItemBase):
    id: int

    model_config = ConfigDict(from_attributes=True)
```

What does from_attributes do?

Without it, Pydantic only accepts dictionaries as input. With it, Pydantic can also read attributes from arbitrary Python objects — crucially, **SQLAlchemy ORM objects**. When we connect to a database in Lesson 8, SQLAlchemy will return objects like this:

```
food = db.query(FoodModel).first()
food.id          # 1
food.name        # "Milk"
food.expiry_date # date(2026, 6, 10)
```

With `from_attributes=True`, FastAPI can pass that object directly to `FoodItemResponse` and it will build correctly:

```
@app.get("/foods/{food_id}", response_model=FoodItemResponse)
def get_food(food_id: int, db: Session = Depends(get_db)):
    food = db.query(FoodModel).filter(FoodModel.id == food_id).first()
    return food  # ← Pydantic reads the attributes directly — no .model_dump() needed
```

Setting it now on `FoodItemResponse` (not `FoodItemCreate` — you never build that from a database object) means no changes needed when we add the database in Lesson 8.

Outside of API requests, you can construct and validate a model directly using `model_validate()`:

```
# From a dictionary
```

```
food = FoodItemCreate.model_validate({
    "name": "Butter",
    "category": "Dairy",
    "expiry_date": "2026-07-01",
})

# From an ORM object (requires from_attributes=True)
food_response = FoodItemResponse.model_validate(orm_object)
```

This is useful in tests, seeding scripts, and anywhere else you need to construct and validate data outside of a request/response cycle.

```
# app/schemas.py
from pydantic import BaseModel, ConfigDict, Field, field_validator, model_validator
from datetime import date
from typing import Optional, Self
from enum import Enum


class CategoryEnum(str, Enum):
    dairy   = "Dairy"
    meat    = "Meat"
    bakery  = "Bakery"
    drinks  = "Drinks"
    produce = "Produce"
    other   = "Other"


class FoodItemBase(BaseModel):
    """Shared fields and validators for all FoodItem schemas."""
    name: str = Field(
        ..., min_length=1, max_length=100,
        description="Name of the food item.", example="Milk",
    )
    category: CategoryEnum = Field(
        ..., description="Food category.", example="Dairy",
    )
    expiry_date: date = Field(
        ..., description="Expiry date (YYYY-MM-DD).", example="2026-06-10",
    )
    notes: Optional[str] = Field(
        default=None, max_length=500,
        description="Optional notes.", example="Full fat, 2 litres",
    )

    @field_validator("name")
    @classmethod
    def normalise_name(cls, v: str) -> str:
        if not v.strip():
            raise ValueError("Name cannot be blank or whitespace only")
        return v.strip().title()

    @field_validator("expiry_date")
    @classmethod
    def expiry_must_be_future(cls, v: date) -> date:
        if v < date.today():
            raise ValueError("Expiry date must be today or in the future")
```

```
        return v


class FoodItemCreate(FoodItemBase):
    """Request body for POST /foods."""
    pass


class FoodItemUpdate(BaseModel):
    """Request body for PATCH /foods/{id} — all fields optional."""
    name:        Optional[str]          = Field(default=None, min_length=1, max_length=100)
    category:    Optional[CategoryEnum] = None
    expiry_date: Optional[date]         = None
    notes:       Optional[str]          = Field(default=None, max_length=500)

    # ... (truncated for readability)
```

---

1. Add `model_config = ConfigDict(from_attributes=True)` to `FoodItemResponse` and then call `model_validate()` on a plain dictionary in a Python shell to confirm it still works correctly:

```
from app.schemas import FoodItemResponse
food = FoodItemResponse.model_validate({
    "id": 99, "name": "Test", "category": "Dairy",
    "expiry_date": "2026-12-01", "notes": None
})
print(food.name)   # "Test"
print(food.id)     # 99
```

2. Add a `purchase_date` field to `FoodItemCreate` (optional, default `None`) and write a `@model_validator(mode="after")` that ensures `expiry_date > purchase_date` when both are provided. Test via `/docs` — submit a body where `expiry_date` is the same day as `purchase_date` and read the 422 response.

3. Test `exclude_unset=True` directly in a Python shell:

```
from app.schemas import FoodItemUpdate
u = FoodItemUpdate(notes="semi-skimmed")
print(u.model_dump())                   # shows name, category, expiry_date as None
print(u.model_dump(exclude_unset=True))  # shows only notes
```

Confirm you understand the difference — this is the mechanism that makes PATCH work correctly.

- **Model inheritance** eliminates field duplication — put shared fields in `FoodItemBase`, inherit in `FoodItemCreate` and `FoodItemResponse`.
- `FoodItemUpdate` has all-optional fields — it's a separate class, not an inheritance shortcut. Validators guard against invalid values even when fields are optional.

- `exclude_unset=True` in `model_dump()` returns only the fields the caller actually sent — essential for correct PATCH behaviour.
- `@model_validator(mode="after")` runs after all fields are set and can enforce cross-field rules. Return `self` when valid.
- `model_config = ConfigDict(from_attributes=True)` on `FoodItemResponse` lets Pydantic build the model from SQLAlchemy ORM objects — set it now, benefit in Lesson 8.
- `model_validate()` constructs and validates a model from a dict or object in your own code — not just from API requests.

In **Lesson 7** we'll build the complete HTTP method surface of the food tracker — POST, PUT, PATCH, and DELETE — putting `FoodItemCreate`, `FoodItemUpdate`, and `FoodItemResponse` to work in a full CRUD interface.

LESSON 07

FastAPI Lesson 7 — HTTP Methods: POST, PUT, PATCH & DELETE

We've been building `GET` endpoints since Lesson 3. In this lesson we complete the CRUD surface of the food tracker — `POST` to create, `PUT` to replace, `PATCH` to partially update, and `DELETE` to remove. We'll also introduce `APIRouter` to keep route code out of `main.py`, and handle 404 errors properly with `HTTPException`.

Before writing code, it's worth being clear about what each method is for:

Method	Purpose	Body?	Typical status
`GET`	Read — fetch one or many	No	200 OK
`POST`	Create — add a new resource	Yes	201 Created
`PUT`	Replace — full update of an existing resource	Yes	200 OK
`PATCH`	Partial update — change only provided fields	Yes	200 OK
`DELETE`	Remove a resource	No	204 No Content

The distinction between `PUT` and `PATCH` matters in practice:

- `PUT` requires the caller to send **all** fields — it replaces the stored record entirely.
- `PATCH` accepts only the fields that are changing — everything else stays as it was.

For the food tracker, `PATCH` is more useful. You rarely want to re-send `name`, `category`, and `expiry_date` just to update `notes`.

So far all routes live in `main.py`. As the project grows this becomes unwieldy. FastAPI's `APIRouter` lets you define routes in separate files and include them in the main app — exactly like Flask blueprints or Django URL includes.

Create app/routers/__init__.py

```
# app/routers/__init__.py
```

(Empty — marks the directory as a Python package.)

Create app/routers/foods.py

```
# app/routers/foods.py
from fastapi import APIRouter

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

- `prefix="/foods"` — every route in this file is automatically prefixed with `/foods`. `@router.get("/{food_id}")` becomes `GET /foods/{food_id}`.
- `tags=["foods"]` — groups all routes under a "foods" section in `/docs`.

Include the router in main.py

```
# app/main.py
from fastapi import FastAPI
from app import config
from app.routers import foods

app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
)

app.include_router(foods.router)
```

Once included, all routes registered on `foods.router` are active in the app.

When a food item ID doesn't exist, the correct response is a `404 Not Found`, not a `200 OK` with an error message. FastAPI provides `HTTPException` for this:

```
from fastapi import HTTPException

raise HTTPException(status_code=404, detail="Food item not found")
```

FastAPI catches this and returns:

```
HTTP/1.1 404 Not Found
{
  "detail": "Food item not found"
}
```

This is much cleaner than returning `{"error": "..."}` from an endpoint typed to return `FoodItemResponse`.

For now, food items are stored in a module-level list. Move this to `routers/foods.py`:

```
# app/routers/foods.py
from datetime import date
from typing import Optional

from fastapi import APIRouter, HTTPException, Path, Query, status

from app.schemas import CategoryEnum, FoodItemCreate, FoodItemResponse, FoodItemUpdate

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

FOODS: list[dict] = [
    {"id": 1, "name": "Milk",         "category": "Dairy",  "expiry_date": "2026-06-10", "notes": None},
    {"id": 2, "name": "Eggs",         "category": "Dairy",  "expiry_date": "2026-06-14", "notes": None},
    {"id": 3, "name": "Chicken",      "category": "Meat",   "expiry_date": "2026-06-08", "notes": None},
    {"id": 4, "name": "Bread",        "category": "Bakery", "expiry_date": "2026-06-09", "notes": None},
    {"id": 5, "name": "Cheddar",      "category": "Dairy",  "expiry_date": "2026-06-20", "notes": None},
    {"id": 6, "name": "Orange Juice", "category": "Drinks", "expiry_date": "2026-06-12", "notes": None},
]

_next_id = 7  # tracks the next ID to assign
```

Move the existing GET routes from `main.py` to `foods.py`. With `prefix="/foods"`, the path arguments shorten:

```
@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category: Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]  = Query(default=None),
    limit: int = Query(default=10, ge=1, le=100),
):
    results = FOODS
    if category:
        results = [f for f in results if f["category"] == category.value]
    if expiring_before:
        results = [f for f in results if f["expiry_date"] <= str(expiring_before)]
    return results[:limit]

@router.get("/{food_id}", response_model=FoodItemResponse)
```

```
def get_food(food_id: int = Path(..., ge=1)):
    for food in FOODS:
        if food["id"] == food_id:
            return food
    raise HTTPException(status_code=404, detail="Food item not found")
```

Note `@router.get("")` (empty string) — with the `/foods` prefix, this registers as `GET /foods`.

```
@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(food: FoodItemCreate):
    global _next_id
    new_food = {"id": _next_id, **food.model_dump(mode="json")}
    FOODS.append(new_food)
    _next_id += 1
    return new_food
```

Key points:

- `status_code=status.HTTP_201_CREATED` — correct HTTP semantics for resource creation. FastAPI uses `200` by default; we override it here.
- `food.model_dump(mode="json")` — `mode="json"` converts Python types like `date` to JSON-serialisable strings. Without it, dates stay as `date` objects in the dictionary.
- The `global _next_id` is a stopgap — in Lesson 8, the database handles ID assignment automatically.

```
@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(food_id: int, food: FoodItemCreate):
    for i, existing in enumerate(FOODS):
        if existing["id"] == food_id:
            updated = {"id": food_id, **food.model_dump(mode="json")}
            FOODS[i] = updated
            return updated
    raise HTTPException(status_code=404, detail="Food item not found")
```

`PUT` uses `FoodItemCreate` as the body — the caller must send all required fields. The stored record is replaced entirely (preserving only the `id`).

```
@router.patch("/{food_id}", response_model=FoodItemResponse)
def update_food(food_id: int, updates: FoodItemUpdate):
    for food in FOODS:
        if food["id"] == food_id:
            update_data = updates.model_dump(exclude_unset=True, mode="json")
```

```
            food.update(update_data)
            return food
    raise HTTPException(status_code=404, detail="Food item not found")
```

`PATCH` uses `FoodItemUpdate` — all fields optional. `exclude_unset=True` ensures only the fields the caller actually sent are merged into the stored record. Everything else stays as it was.

```
@router.delete("/{food_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_food(food_id: int = Path(..., ge=1)):
    for i, food in enumerate(FOODS):
        if food["id"] == food_id:
            FOODS.pop(i)
            return
    raise HTTPException(status_code=404, detail="Food item not found")
```

Key points:

- `status_code=204` — "No Content" is the correct status for a successful delete. The response body is empty.
- **No** `response_model` — a 204 response has no body, so there's nothing to model.
- Return with bare `return` (or `None`) — FastAPI sends the 204 and no body.

```
# app/routers/foods.py
from datetime import date
from typing import Optional

from fastapi import APIRouter, HTTPException, Path, Query, status

from app.schemas import CategoryEnum, FoodItemCreate, FoodItemResponse, FoodItemUpdate

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

FOODS: list[dict] = [
    {"id": 1, "name": "Milk",         "category": "Dairy",  "expiry_date": "2026-06-10", "notes": None},
    {"id": 2, "name": "Eggs",         "category": "Dairy",  "expiry_date": "2026-06-14", "notes": None},
    {"id": 3, "name": "Chicken",      "category": "Meat",   "expiry_date": "2026-06-08", "notes": None},
    {"id": 4, "name": "Bread",        "category": "Bakery", "expiry_date": "2026-06-09", "notes": None},
    {"id": 5, "name": "Cheddar",      "category": "Dairy",  "expiry_date": "2026-06-20", "notes": None},
    {"id": 6, "name": "Orange Juice", "category": "Drinks", "expiry_date": "2026-06-12", "notes": None},
]

_next_id = 7


def _find_food(food_id: int) -> dict:
    """Return the food dict for food_id, or raise 404."""
    for food in FOODS:
        if food["id"] == food_id:
            return food
```

```
    raise HTTPException(status_code=404, detail="Food item not found")


@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category: Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]  = Query(default=None),
    limit: int = Query(default=10, ge=1, le=100),
):
    results = FOODS
    if category:
        results = [f for f in results if f["category"] == category.value]
    if expiring_before:
        results = [f for f in results if f["expiry_date"] <= str(expiring_before)]
    return results[:limit]


@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(food: FoodItemCreate):
    global _next_id
    new_food = {"id": _next_id, **food.model_dump(mode="json")}
    FOODS.append(new_food)
    _next_id += 1
    return new_food


@router.get("/{food_id}", response_model=FoodItemResponse)
def get_food(food_id: int = Path(..., ge=1)):
    return _find_food(food_id)


@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(food_id: int, food: FoodItemCreate):
    # ... (truncated for readability)
```

```
# app/main.py
from fastapi import FastAPI
from app import config
from app.routers import foods

app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
)

app.include_router(foods.router)
```

`main.py` is now a thin entry point — application setup and router registration only.

Start the server and open `/docs`. You'll see all six endpoints grouped under **foods**:

```
GET    /foods
POST   /foods
GET    /foods/{food_id}
PUT    /foods/{food_id}
PATCH  /foods/{food_id}
DELETE /foods/{food_id}
```

Work through each in sequence:

1. **GET /foods** — confirm the seed data is returned
2. **POST /foods** — create a new item; confirm id is 7 and it appears in `GET /foods`
3. **GET /foods/7** — retrieve the item you just created
4. **PUT /foods/7** — replace it entirely with different fields
5. **PATCH /foods/7** — change only `notes`; confirm name and category are unchanged
6. **DELETE /foods/7** — delete it; confirm `GET /foods/7` returns 404 afterwards

---

1. Test that `PUT /foods/{id}` enforces required fields. Send a PUT body missing `category` and confirm you get a `422`. Compare to `PATCH`, where omitting `category` is perfectly valid.

2. Add a `GET /foods/expiring-soon` route to `foods.py` that returns all items expiring within the next 7 days. Remember route order — declare it **before** `GET /foods/{food_id}` to avoid FastAPI treating `expiring-soon` as a path parameter.

```
from datetime import date, timedelta

@router.get("/expiring-soon", response_model=list[FoodItemResponse])
def get_expiring_soon():
    cutoff = date.today() + timedelta(days=7)
    return [f for f in FOODS if f["expiry_date"] <= str(cutoff)]
```

3. Notice that `DELETE /foods/999` (a non-existent ID) returns a `404`, but the current implementation loops through `FOODS` twice — once in the loop and then the `HTTPException` at the end. Refactor `delete_food` to use `_find_food()` as a helper first, then find the index separately. (Hint: look at how `replace_food` handles this.)

- `POST` creates resources and returns `201 Created`. `PUT` replaces. `PATCH` partially updates. `DELETE` removes and returns `204 No Content`.
- `APIRouter` moves routes out of `main.py` into feature-specific files. Register with `app.include_router()`. The `prefix` argument applies to every route in the router.

- `HTTPException` raises structured HTTP errors. Always raise a 404 rather than returning an error dict from an endpoint with a typed response model.
- `status.HTTP_201_CREATED` and `status.HTTP_204_NO_CONTENT` from `fastapi` are named constants — use them instead of raw integers.
- `mode="json"` in `model_dump()` converts Python types (like `date`) to JSON-serialisable values, which is needed when storing to a plain dict.

In **Lesson 8** we'll swap the in-memory `FOODS` list for a real SQLite database using SQLAlchemy, and introduce the `database.py` and `models.py` files. The `FoodItemResponse` model is already prepared for this — `from_attributes=True` means no schema changes needed when we switch.

LESSON 08

FastAPI Lesson 8 — Intro to SQLAlchemy & SQLite

The in-memory `FOODS` list has served its purpose — but every time the server restarts, all data disappears. In this lesson we replace it with a real SQLite database, managed through SQLAlchemy. We'll create `database.py` (the connection layer), `models.py` (the table definition), wire up a database session dependency, and update the router to use actual SQL queries. By the end, the food tracker persists data across restarts.

```
pip install sqlalchemy
```

SQLite is part of Python's standard library — no separate installation needed. Add `sqlalchemy` to `requirements.txt`:

```
fastapi
uvicorn[standard]
pydantic[email]
python-dotenv
sqlalchemy
```

From this lesson onwards, the word "model" is overloaded. It's important to keep the two meanings separate:

Term	File	Inherits from	Purpose
SQLAlchemy model	`app/models.py`	`Base` (SQLAlchemy)	Maps a Python class to a database table
Pydantic schema	`app/schemas.py`	`BaseModel` (Pydantic)	Validates and serialises API request/response data

SQLAlchemy models describe *how data is stored*. Pydantic schemas describe *how data looks on the wire*. FastAPI bridges the two: it reads from the database using SQLAlchemy, then serialises to JSON using Pydantic.

```python
# app/database.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, DeclarativeBase

DATABASE_URL = "sqlite:///./food_tracker.db"

engine = create_engine(
    DATABASE_URL,
    connect_args={"check_same_thread": False},  # SQLite-specific — required for FastAPI
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)


class Base(DeclarativeBase):
    pass
```

What each piece does

`engine` — the connection to the database file. SQLAlchemy creates `food_tracker.db` in the project root on first run.

`connect_args={"check_same_thread": False}` — SQLite's default behaviour only allows the same thread to use a connection. FastAPI uses multiple threads, so this restriction must be lifted.

`SessionLocal` — a factory for database sessions. Each request gets its own session. `autocommit=False` means changes aren't written until you call `db.commit()`. `autoflush=False` means SQLAlchemy doesn't automatically synchronise the session state before every query.

`Base` — the parent class for all SQLAlchemy models. Any class that inherits from it and defines `__tablename__` becomes a mapped database table.

```python
# app/models.py
from sqlalchemy import Column, Date, Integer, String
from app.database import Base


class FoodItem(Base):
    __tablename__ = "food_items"

    id          = Column(Integer, primary_key=True, index=True)
    name        = Column(String(100), nullable=False)
    category    = Column(String(50),  nullable=False)
    expiry_date = Column(Date,        nullable=False)
    notes       = Column(String(500), nullable=True)
```

Each attribute is a `Column`. The type (`Integer`, `String`, `Date`) maps to the equivalent SQL type. `primary_key=True` makes `id` the auto-incrementing primary key — SQLite assigns this automatically on

insert, which is why we no longer need `_next_id`.

Naming: FoodItem vs FoodItemCreate

The SQLAlchemy model is also called `FoodItem`. To avoid a name collision when both are imported in the same file, import the SQLAlchemy model with an alias:

```
from app.models import FoodItem as FoodItemModel
```

This is the convention used in the router below.

SQLAlchemy creates the database tables from the model definitions using `Base.metadata.create_all()`. Call this once when the app starts. FastAPI's `lifespan` context manager is the right place:

```
# app/main.py
from contextlib import asynccontextmanager
from fastapi import FastAPI
from app import config
from app.database import engine, Base
from app.routers import foods


@asynccontextmanager
async def lifespan(app: FastAPI):
    # Runs on startup — creates tables if they don't exist
    Base.metadata.create_all(bind=engine)
    yield
    # Runs on shutdown — add cleanup here if needed


app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
    lifespan=lifespan,
)

app.include_router(foods.router)
```

`create_all()` is safe to call on every startup — it checks whether each table already exists before creating it. It does **not** modify existing tables if the schema changes. (Schema migrations are covered separately, and Alembic is the standard tool for them — but `create_all()` is sufficient while the project is in development.)

Database sessions must be opened before a request is handled and closed afterwards, regardless of whether the request succeeded or raised an error. A FastAPI dependency with `yield` handles this cleanly:

```python
# app/database.py (add to the bottom)
from typing import Generator
from sqlalchemy.orm import Session


def get_db() -> Generator[Session, None, None]:
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

`yield` makes this a *generator dependency*. FastAPI runs the code before `yield` to set up the session, injects the yielded value into the endpoint function, runs the endpoint, then runs the code after `yield` (the `finally` block) to close the session — even if the endpoint raised an exception.

Any endpoint that needs the database declares it as a parameter:

```python
from fastapi import Depends
from sqlalchemy.orm import Session
from app.database import get_db

@router.get("/{food_id}")
def get_food(food_id: int, db: Session = Depends(get_db)):
    ...
```

`Depends(get_db)` tells FastAPI to call `get_db()`, take the yielded `Session`, and pass it as `db`. This is **dependency injection** — the endpoint doesn't create or manage the session itself.

Remove the `FOODS` list and `_next_id` counter. All data operations now go through `db`:

GET — List

```python
@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category:        Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]         = Query(default=None),
    limit:           int                    = Query(default=10, ge=1, le=100),
    db:              Session                = Depends(get_db),
):
    query = db.query(FoodItemModel)
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    if expiring_before:
        query = query.filter(FoodItemModel.expiry_date <= expiring_before)
    return query.limit(limit).all()
```

`db.query(FoodItemModel)` starts a SELECT query. `.filter()` adds WHERE clauses. `.limit()` adds LIMIT. `.all()` executes and returns a list of `FoodItemModel` instances.

`FoodItemResponse` has `from_attributes=True` — so FastAPI passes the ORM objects directly to Pydantic, which reads their attributes (`.name`, `.expiry_date`, etc.) to build the response. No manual

conversion needed.

GET — Single

```
@router.get("/{food_id}", response_model=FoodItemResponse)
def get_food(food_id: int = Path(..., ge=1), db: Session = Depends(get_db)):
    food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return food
```

`.first()` returns the first matching row, or `None` if there are no matches.

POST — Create

```
@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(food: FoodItemCreate, db: Session = Depends(get_db)):
    db_food = FoodItemModel(**food.model_dump())
    db.add(db_food)
    db.commit()
    db.refresh(db_food)
    return db_food
```

- `FoodItemModel(**food.model_dump())` — create a SQLAlchemy model instance from the validated Pydantic data. Note: no `mode="json"` here — we want native Python types (`date`, not a string) so SQLAlchemy stores them correctly.
- `db.add(db_food)` — stage the object for insertion.
- `db.commit()` — write to the database. After this, `db_food.id` is populated by SQLite.
- `db.refresh(db_food)` — reload the object from the database to pick up the server-assigned `id` and any defaults.

PUT — Full Replacement

```
@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(food_id: int, food: FoodItemCreate, db: Session = Depends(get_db)):
    db_food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if db_food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    for field, value in food.model_dump().items():
        setattr(db_food, field, value)
    db.commit()
    db.refresh(db_food)
    return db_food
```

`setattr` updates each attribute on the existing ORM object. `db.commit()` writes all changes.

PATCH — Partial Update

```
@router.patch("/{food_id}", response_model=FoodItemResponse)
def update_food(food_id: int, updates: FoodItemUpdate, db: Session = Depends(get_db)):
    db_food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if db_food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    for field, value in updates.model_dump(exclude_unset=True).items():
        setattr(db_food, field, value)
```

```
    db.commit()
    db.refresh(db_food)
    return db_food
```

Same pattern as PUT but using `exclude_unset=True` — only the fields the caller sent are iterated and updated.

DELETE — Remove

```
@router.delete("/{food_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_food(food_id: int = Path(..., ge=1), db: Session = Depends(get_db)):
    db_food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if db_food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    db.delete(db_food)
    db.commit()
```

`db.delete(db_food)` marks the object for deletion. `db.commit()` executes the DELETE.

```
# app/routers/foods.py
from datetime import date
from typing import Optional

from fastapi import APIRouter, Depends, HTTPException, Path, Query, status
from sqlalchemy.orm import Session

from app.database import get_db
from app.models import FoodItem as FoodItemModel
from app.schemas import CategoryEnum, FoodItemCreate, FoodItemResponse, FoodItemUpdate

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


@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category:        Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]         = Query(default=None),
    limit:           int                    = Query(default=10, ge=1, le=100),
    db:              Session                = Depends(get_db),
):
    query = db.query(FoodItemModel)
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    if expiring_before:
        query = query.filter(FoodItemModel.expiry_date <= expiring_before)
    return query.limit(limit).all()


@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(food: FoodItemCreate, db: Session = Depends(get_db)):
    db_food = FoodItemModel(**food.model_dump())
    db.add(db_food)
    db.commit()
    db.refresh(db_food)
    return db_food
```

```python
@router.get("/expiring-soon", response_model=list[FoodItemResponse])
def get_expiring_soon(
    days: int = Query(default=7, ge=1, le=30),
    db: Session = Depends(get_db),
):
    from datetime import timedelta
    cutoff = date.today() + timedelta(days=days)
    return db.query(FoodItemModel).filter(FoodItemModel.expiry_date <= cutoff).all()


@router.get("/{food_id}", response_model=FoodItemResponse)
def get_food(food_id: int = Path(..., ge=1), db: Session = Depends(get_db)):
    food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return food


@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(food_id: int, food: FoodItemCreate, db: Session = Depends(get_db)):
    db_food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if db_food is None:
    # ... (truncated for readability)
```

```
food_tracker/
■■■ venv/
■■■ app/
■   ■■■ __init__.py
■   ■■■ main.py          ← updated (lifespan + create_all)
■   ■■■ config.py
■   ■■■ database.py      ← NEW
■   ■■■ models.py        ← NEW
■   ■■■ schemas.py       ← unchanged
■   ■■■ routers/
■       ■■■ __init__.py
■       ■■■ foods.py     ← updated (DB session replaces FOODS list)
■■■ food_tracker.db      ← created on first run
■■■ tests/
■■■ .env
■■■ .gitignore
■■■ requirements.txt
```

---

1. Start the server and use `/docs` to create three food items via POST. Stop the server with `Ctrl+C` and restart it. Confirm the items are still there — this is the key difference from the in-memory list.

2. Open `food_tracker.db` using DB Browser for SQLite (free download at https://sqlitebrowser.org) or any SQLite viewer. Browse the `food_items` table. Confirm the rows match what you created via the API.

3. Add a `GET /foods/expiring-soon` endpoint that accepts a `days` query parameter (default 7, max 30) and returns all items expiring within that many days. The endpoint is already in the complete `foods.py` above — read it, understand the query, then write it yourself from memory before copying.

4. (Advanced) What happens if you add a new column to the `FoodItem` SQLAlchemy model and restart the server? Try adding `quantity = Column(Integer, nullable=True, default=1)` and confirm that `create_all()` does **not** add the column to the existing table. This is why Alembic (database migrations) exists — the topic of Lesson 9.

- **SQLAlchemy model** (in `models.py`) maps a Python class to a database table. **Pydantic schema** (in `schemas.py`) validates API data. They are separate — don't confuse them.
- `database.py` contains the engine (connection), `SessionLocal` (session factory), and `Base` (parent for ORM models).
- `connect_args={"check_same_thread": False}` is required for SQLite with FastAPI's threaded request handling.
- `get_db()` is a generator dependency that yields a session per request and closes it in a `finally` block — ensuring cleanup even on errors.
- `db.add()` → `db.commit()` → `db.refresh()` is the create pattern. Commit writes to disk; refresh reloads the object so the server-assigned `id` is available.
- `from_attributes=True` on `FoodItemResponse` (set in Lesson 6) means Pydantic reads from ORM object attributes directly — no `.model_dump()` or manual conversion needed when returning from endpoints.
- `Base.metadata.create_all()` creates tables on startup if they don't exist. It is not a migration tool — it won't modify existing tables.

In **Lesson 9** we'll introduce Alembic — the migration tool that handles schema changes to existing tables. We'll use it to add columns to `food_items` without losing existing data.

LESSON 09

FastAPI Lesson 9 — Database Migrations with Alembic

In Lesson 8 you discovered that `Base.metadata.create_all()` only creates missing tables — it can't add a column to a table that already exists. In a real project, schemas evolve: new columns are added, old ones are removed, types change. **Alembic** is the migration tool for SQLAlchemy. It tracks every schema change as a versioned script, letting you upgrade or roll back the database structure without losing data.

Imagine the food tracker is deployed and has 500 rows in `food_items`. You decide to add a `quantity` column. Without Alembic:

- Deleting and recreating the database loses all 500 rows.
- Manually writing `ALTER TABLE` SQL is error-prone and hard to reproduce across environments.

With Alembic, you run two commands:

```
alembic revision --autogenerate -m "add quantity column"
alembic upgrade head
```

Alembic inspects the difference between the current database schema and the SQLAlchemy models, generates an `ALTER TABLE` script automatically, and applies it. The existing rows are preserved.

```
pip install alembic
```

Add to `requirements.txt`:

```
alembic
```

From the project root (`food_tracker/`):

```
alembic init alembic
```

This creates:

```
food_tracker/
■■■ alembic/
■   ■■■ env.py           ← migration environment (configure this)
■   ■■■ script.py.mako   ← template for generated migration scripts
■   ■■■ versions/        ← migration scripts go here
■■■ alembic.ini          ← Alembic configuration (configure this)
```

Open `alembic.ini`. Find the line:

```
sqlalchemy.url = driver://user:pass@localhost/dbname
```

Replace it with the SQLite URL used in `database.py`:

```
sqlalchemy.url = sqlite:///./food_tracker.db
```

> **Note:** In production with environment variables, you'd leave this blank and set the URL dynamically in `env.py` instead (shown below in the advanced note).

The generated `env.py` needs to know about the SQLAlchemy models so it can compare the current database state against them. Make two changes:

1. Import `Base` from your app:

Near the top of `env.py`, after the existing imports:

```
# alembic/env.py — add these two lines after the existing imports
import sys, os
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))

from app.database import Base  # noqa: E402
```

The `sys.path.insert` ensures Python can find the `app` package when Alembic runs from the project root.

2. Set `target_metadata`:

Find this line (already in the generated file):

```
target_metadata = None
```

Replace it with:

```
target_metadata = Base.metadata
```

This tells Alembic to compare the database against all SQLAlchemy models that inherit from `Base` — which is how `--autogenerate` works.

Before adding any new columns, create a migration that represents the current schema:

```
alembic revision --autogenerate -m "initial migration"
```

Alembic compares the current database (empty, or already has a `food_items` table from `create_all()`) against the models and generates a script in `alembic/versions/`. The filename looks like:

```
alembic/versions/a1b2c3d4e5f6_initial_migration.py
```

Open the generated file — it will contain `upgrade()` and `downgrade()` functions:

```
def upgrade() -> None:
    op.create_table(
        "food_items",
        sa.Column("id", sa.Integer(), nullable=False),
        sa.Column("name", sa.String(length=100), nullable=False),
        sa.Column("category", sa.String(length=50), nullable=False),
        sa.Column("expiry_date", sa.Date(), nullable=False),
        sa.Column("notes", sa.String(length=500), nullable=True),
        sa.PrimaryKeyConstraint("id"),
    )
    op.create_index(op.f("ix_food_items_id"), "food_items", ["id"], unique=False)


def downgrade() -> None:
    op.drop_index(op.f("ix_food_items_id"), table_name="food_items")
    op.drop_table("food_items")
```

Alembic generated this from your model definition. You didn't write any SQL.

```
alembic upgrade head
```

`head` means "apply all pending migrations up to the latest one". Alembic:

1. Checks whether `food_tracker.db` already has an `alembic_version` table. If not, it creates it.
2. Reads the version stored there (if any) and compares to the migration chain.
3. Runs every pending `upgrade()` function in sequence.
4. Writes the new version hash to `alembic_version`.

> **If you already ran `create_all()` in Lesson 8:** the `food_items` table already exists. Alembic will detect this and either skip the `CREATE TABLE` or raise an error. The safest approach is to delete `food_tracker.db` and let Alembic create a clean database. In a real project you'd stamp the existing database with `alembic stamp head` instead.

Alembic adds a small table to your database called `alembic_version`. It holds one row — the hash of the last applied migration:

```
alembic_version
┌──────────────────────────────────────────┐
│ version_num                              │
├──────────────────────────────────────────┤
│ a1b2c3d4e5f6                             │
└──────────────────────────────────────────┘
```

When you run `alembic upgrade head`, Alembic reads this value, finds the migration script with that hash, follows the chain of `down_revision` links to determine what's pending, and runs them in order.

This is the core Alembic workflow you'll use repeatedly. We'll add a `quantity` field to `food_items`.

Step 1 — Update the SQLAlchemy model

```python
# app/models.py
class FoodItem(Base):
    __tablename__ = "food_items"

    id          = Column(Integer, primary_key=True, index=True)
    name        = Column(String(100), nullable=False)
    category    = Column(String(50),  nullable=False)
    expiry_date = Column(Date,        nullable=False)
    notes       = Column(String(500), nullable=True)
    quantity    = Column(Integer,     nullable=True, default=1)  # ← NEW
```

Step 2 — Update the Pydantic schemas

Add `quantity` to `FoodItemBase` (so `FoodItemCreate` and `FoodItemResponse` both pick it up) and `FoodItemUpdate`:

```python
# app/schemas.py — add to FoodItemBase
quantity: Optional[int] = Field(default=1, ge=1, description="Number of units.")

# app/schemas.py — add to FoodItemUpdate
quantity: Optional[int] = Field(default=None, ge=1)
```

Step 3 — Generate the migration

```
alembic revision --autogenerate -m "add quantity column"
```

Alembic generates a new file in `versions/`. Inspect it — it should contain:

```
def upgrade() -> None:
    op.add_column("food_items", sa.Column("quantity", sa.Integer(), nullable=True))


def downgrade() -> None:
    op.drop_column("food_items", "quantity")
```

Always read the generated migration before applying it. Autogenerate is good but not perfect — it can miss some changes (like server defaults or check constraints) and should be treated as a first draft.

Step 4 — Apply

```
alembic upgrade head
```

The `quantity` column is now in the database. Existing rows get `NULL` for `quantity` (because it's nullable). The 500 hypothetical rows are untouched.

Now that Alembic manages the schema, `create_all()` in `main.py` is redundant — and slightly dangerous, because it could create tables that bypass Alembic's version tracking. Remove the lifespan block or simplify it:

```
# app/main.py — simplified (no create_all)
from contextlib import asynccontextmanager
from fastapi import FastAPI
from app import config
from app.routers import foods


@asynccontextmanager
async def lifespan(app: FastAPI):
    yield  # startup/shutdown hooks go here if needed


app = FastAPI(
    title=config.APP_NAME,
    version=config.APP_VERSION,
    description="Track food items and their expiration dates.",
    lifespan=lifespan,
)

app.include_router(foods.router)
```

Database schema is now managed entirely by Alembic. Run `alembic upgrade head` before starting the server in any environment.

To undo the last migration:

```
alembic downgrade -1
```

`-1` means "go back one step". This runs the `downgrade()` function in the most recent migration — in our case, `op.drop_column("food_items", "quantity")`. The column is removed and the version table is updated.

To downgrade all the way to the beginning:

```
alembic downgrade base
```

Command	What it does
`alembic revision --autogenerate -m "message"`	Generate a migration by comparing models to the database
`alembic revision -m "message"`	Generate an empty migration script (write `upgrade()`/`downgrade()` manually)
`alembic upgrade head`	Apply all pending migrations
`alembic upgrade +1`	Apply one migration
`alembic downgrade -1`	Undo the last migration
`alembic downgrade base`	Undo all migrations
`alembic current`	Show the current database revision
`alembic history`	List all migrations in order
`alembic show <rev>`	Show the script for a specific revision
`alembic stamp head`	Mark the database as up to date without running migrations

---

```
food_tracker/
■■■ alembic/
■   ■■■ env.py                    ← configured with Base.metadata
■   ■■■ script.py.mako
■   ■■■ versions/
■       ■■■ a1b2c3d4_initial_migration.py
■       ■■■ b2c3d4e5_add_quantity_column.py
■■■ alembic.ini                   ← sqlalchemy.url set
■■■ venv/
■■■ app/
```

```
■   ■■■ main.py                    ← create_all() removed
■   ■■■ config.py
■   ■■■ database.py
■   ■■■ models.py                  ← quantity column added
■   ■■■ schemas.py                 ← quantity field added
■   ■■■ routers/
■       ■■■ foods.py
■■■ food_tracker.db
■■■ .env
■■■ requirements.txt
```

---

1. Run through the full migration setup from scratch. Delete `food_tracker.db`, run `alembic upgrade head`, start the server, and confirm the API works as before. Check `food_tracker.db` with DB Browser — you should see both `food_items` and `alembic_version` tables.

2. Add the `quantity` migration. Follow the four steps above: update `models.py`, update `schemas.py`, run `alembic revision --autogenerate -m "add quantity column"`, inspect the generated file, then run `alembic upgrade head`. Confirm via `/docs` that `POST /foods` now accepts a `quantity` field and stores it correctly.

3. Practise downgrading. Run `alembic downgrade -1`. Open DB Browser and confirm the `quantity` column is gone. Run `alembic upgrade head` to reapply. This is the rollback drill you'd run in an incident.

4. Run `alembic history` and confirm both migrations appear in the correct order. Run `alembic current` and confirm it shows the latest revision hash.

- `create_all()` **creates tables; Alembic migrates them.** Use Alembic for any project where the schema will evolve.
- `alembic init alembic` creates the migration directory. `env.py` must be configured with `target_metadata = Base.metadata` and the correct `sys.path`.
- `--autogenerate` compares SQLAlchemy models to the live database and generates the migration script. Always inspect the output before applying.
- **Every migration has** `upgrade()` **and** `downgrade()` functions. Both should be correct — you'll rely on `downgrade()` in production incidents.
- `alembic upgrade head` applies all pending migrations. Run this before starting the server in every environment (dev, staging, production).
- The `alembic_version` table tracks the current schema version. Never edit it manually.
- Remove `create_all()` from `main.py` once Alembic is set up — it can create tables outside Alembic's tracking.

In **Lesson 10** we'll extract the database query logic from the router into a dedicated **CRUD layer** — plain functions in a `crud.py` file. This keeps the router thin (HTTP concerns only) and makes the database operations independently testable.

LESSON 10

FastAPI Lesson 10 — The CRUD Layer: Separating Database Logic from Routes

Looking at `app/routers/foods.py` after Lesson 8, each endpoint contains a mix of two distinct concerns: **HTTP handling** (status codes, request bodies, response models, 404 errors) and **database operations** (queries, commits, refreshes). These should live in separate places. This lesson extracts all database logic into a dedicated `app/crud.py` file, leaving the router thin and focused purely on HTTP concerns. The result is code that is easier to read, easier to test, and easier to reuse.

Here is the current PATCH endpoint:

```
@router.patch("/{food_id}", response_model=FoodItemResponse)
def update_food(food_id: int, updates: FoodItemUpdate, db: Session = Depends(get_db)):
    db_food = db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()
    if db_food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    for field, value in updates.model_dump(exclude_unset=True).items():
        setattr(db_food, field, value)
    db.commit()
    db.refresh(db_food)
    return db_food
```

The function is simultaneously:

- A FastAPI route handler (decorator, response model, Depends)
- A database query layer (SQLAlchemy query, filter, commit, refresh)
- An HTTP error handler (HTTPException)

This is fine for a small project. But as the codebase grows — more routes, background tasks, tests, possibly a CLI — the database logic needs to be callable from places other than route handlers. A CRUD layer solves this by making each database operation a plain Python function that takes a session and returns data.

The convention (used in the official FastAPI docs and most real codebases) is:

```
app/
■■■ crud.py       ← database operations as plain functions
■■■ routers/
    ■■■ foods.py  ← HTTP handlers that call crud functions
```

Each function in `crud.py`:

- Accepts a `Session` as its first argument
- Performs one database operation
- Returns an ORM object (or `None`, or a list)
- **Never raises** `HTTPException` — that's the router's job

The router:

- Calls the appropriate crud function
- Checks the return value
- Raises `HTTPException` if needed
- Returns the result with the correct response model and status code

---

```python
# app/crud.py
from datetime import date
from typing import Optional

from sqlalchemy.orm import Session

from app.models import FoodItem as FoodItemModel
from app.schemas import CategoryEnum, FoodItemCreate, FoodItemUpdate


def get_food(db: Session, food_id: int) -> FoodItemModel | None:
    """Return a single FoodItem by id, or None if not found."""
    return db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()


def get_foods(
    db: Session,
    *,
    category: Optional[CategoryEnum] = None,
    expiring_before: Optional[date] = None,
    limit: int = 10,
) -> list[FoodItemModel]:
    """Return a filtered, limited list of FoodItems."""
    query = db.query(FoodItemModel)
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    if expiring_before:
        query = query.filter(FoodItemModel.expiry_date <= expiring_before)
    return query.limit(limit).all()


def get_expiring_soon(db: Session, *, days: int = 7) -> list[FoodItemModel]:
    """Return items expiring within `days` days from today."""
    from datetime import timedelta
    cutoff = date.today() + timedelta(days=days)
    return db.query(FoodItemModel).filter(FoodItemModel.expiry_date <= cutoff).all()
```

```
def create_food(db: Session, food: FoodItemCreate) -> FoodItemModel:
    """Insert a new FoodItem and return it with its server-assigned id."""
    db_food = FoodItemModel(**food.model_dump())
    db.add(db_food)
    db.commit()
    db.refresh(db_food)
    return db_food


def replace_food(db: Session, food_id: int, food: FoodItemCreate) -> FoodItemModel | None:
    """Replace all fields of an existing FoodItem. Returns None if not found."""
    db_food = get_food(db, food_id)
    if db_food is None:
        return None
    for field, value in food.model_dump().items():
        setattr(db_food, field, value)
    db.commit()
    db.refresh(db_food)
    return db_food


def update_food(db: Session, food_id: int, updates: FoodItemUpdate) -> FoodItemModel | None:
    # ... (truncated for readability)
```

Key design decisions

`get_food` **returns** `None`**, not a 404.** The crud function doesn't know or care that it's being called from an HTTP context. The router is responsible for deciding that a `None` result should become a 404.

`delete_food` **returns** `bool`**.** Again — no HTTP exceptions. The router checks the return value and raises `HTTPException` if it's `False`.

`replace_food` **and** `update_food` **return** `None` **on missing.** Same principle: the crud function signals "not found" via `None`, and the router translates that into a 404.

Keyword-only arguments with `*`**.** Functions like `get_foods` use `*` to make `category`, `expiring_before`, and `limit` keyword-only. This prevents accidentally passing them positionally and makes call sites more readable.

With the crud layer in place, the router becomes a clean mapping of HTTP requests to crud calls:

```
# app/routers/foods.py
from datetime import date
from typing import Optional

from fastapi import APIRouter, Depends, HTTPException, Path, Query, status
from sqlalchemy.orm import Session

from app import crud
from app.database import get_db
from app.schemas import CategoryEnum, FoodItemCreate, FoodItemResponse, FoodItemUpdate

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

```python
@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category:        Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]         = Query(default=None),
    limit:           int                    = Query(default=10, ge=1, le=100),
    db:              Session                = Depends(get_db),
):
    return crud.get_foods(db, category=category, expiring_before=expiring_before, limit=limit)


@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(food: FoodItemCreate, db: Session = Depends(get_db)):
    return crud.create_food(db, food)


@router.get("/expiring-soon", response_model=list[FoodItemResponse])
def get_expiring_soon(
    days: int = Query(default=7, ge=1, le=30),
    db: Session = Depends(get_db),
):
    return crud.get_expiring_soon(db, days=days)


@router.get("/{food_id}", response_model=FoodItemResponse)
def get_food(food_id: int = Path(..., ge=1), db: Session = Depends(get_db)):
    food = crud.get_food(db, food_id)
    if food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return food


@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(food_id: int, food: FoodItemCreate, db: Session = Depends(get_db)):
    result = crud.replace_food(db, food_id, food)
    if result is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return result


@router.patch("/{food_id}", response_model=FoodItemResponse)
def update_food(food_id: int, updates: FoodItemUpdate, db: Session = Depends(get_db)):
    result = crud.update_food(db, food_id, updates)
    if result is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return result

    # ... (truncated for readability)
```

Each route handler is now three to six lines. The logic is in `crud.py`; the router handles HTTP conventions.

---

1. Testability

You can test the entire database layer without FastAPI:

```python
# In a test file — no HTTP client needed
from app import crud
```

```
from app.schemas import FoodItemCreate

def test_create_food(db_session):
    food_in = FoodItemCreate(
        name="Test Milk",
        category="Dairy",
        expiry_date="2026-12-01",
    )
    food = crud.create_food(db_session, food_in)
    assert food.id is not None
    assert food.name == "Test Milk"
```

Lesson 16 covers testing in full — but the key point is that `crud.py` functions are plain Python, testable with any test runner.

2. Reusability

Background tasks, CLI scripts, and scheduled jobs can call `crud.get_expiring_soon()` directly without going through the HTTP layer:

```
# In a background task (Lesson 13)
from app import crud

def send_expiry_notifications(db: Session):
    items = crud.get_expiring_soon(db, days=3)
    for item in items:
        # send notification...
```

3. Readable routes

The router now reads almost like a list of HTTP operations:

- `GET /foods` → `crud.get_foods()`
- `POST /foods` → `crud.create_food()`
- `GET /foods/{id}` → `crud.get_food()` + 404 check
- `PATCH /foods/{id}` → `crud.update_food()` + 404 check
- `DELETE /foods/{id}` → `crud.delete_food()` + 404 check

```
food_tracker/
├── alembic/
├── alembic.ini
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── config.py
│   ├── database.py
│   ├── models.py
│   ├── schemas.py
│   ├── crud.py          ← NEW: all database operations
│   ├── routers/
│   │   ├── __init__.py
│   │   └── foods.py     ← updated: thin HTTP handlers only
```

```
■■■ food_tracker.db
■■■ requirements.txt
```

---

1. Read through both `crud.py` and the updated `foods.py` side by side. For each route handler in `foods.py`, trace exactly which crud function is called and what it returns. Confirm you understand why the router checks `if result is None` before returning — and what would happen if you removed that check.

2. Add a `count_foods` function to `crud.py` that returns the total number of food items in the database, optionally filtered by category:

```
def count_foods(db: Session, *, category: Optional[CategoryEnum] = None) -> int:
    query = db.query(FoodItemModel)
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    return query.count()
```

Then add a `GET /foods/count` endpoint in the router that calls it and returns `{"count": n}`. Remember route order — declare it before `GET /foods/{food_id}`.

3. Call a crud function directly from the Python REPL to confirm it works independently of FastAPI:

```
from app.database import SessionLocal
from app import crud

db = SessionLocal()
foods = crud.get_foods(db, limit=5)
for f in foods:
    print(f.name, f.expiry_date)
db.close()
```

- `crud.py` **contains database operations; the router contains HTTP handling.** Neither should know the other's language — crud functions don't raise `HTTPException`; route handlers don't write raw SQLAlchemy queries.
- **Crud functions return `None` or `bool` to signal "not found".** The router translates these signals into `HTTPException(404)`.
- **Keyword-only arguments (`*`) in crud functions** make call sites self-documenting and prevent positional argument mistakes.
- **Testability is the main benefit.** Crud functions are plain Python — no test client, no HTTP headers, no JSON parsing required to test them.
- **Reusability is the second benefit.** Background tasks, CLI scripts, and scheduled jobs can call crud functions directly.

In **Lesson 11** we'll look at FastAPI's dependency injection system in depth — building reusable dependencies for things like pagination parameters, the current user, and feature flags, keeping the router signatures clean as complexity grows.

LESSON 11

FastAPI Lesson 11 — Dependency Injection in Depth

FastAPI's `Depends()` system does more than just hand the router a database session. Any callable — a function, a class, even another dependency — can be declared as a dependency, and FastAPI will resolve, cache, and inject it automatically. This lesson builds a set of reusable dependencies that clean up the router signatures you've accumulated: pagination parameters, a "current user" stub, and a feature-flag guard. By the end, each route handler declares only the parameters it actually uses.

`Depends()`

When FastAPI sees `Depends(some_callable)` in a route signature, it:

1. Inspects `some_callable`'s own parameters.
2. Resolves those parameters — from the request, from other dependencies, or from `Depends()` references inside them.
3. Calls `some_callable` with the resolved values.
4. Injects the return value into the route handler.

This happens before the route body runs. If any dependency raises `HTTPException`, the route body is never called.

```
from fastapi import Depends, FastAPI

def get_x() -> int:
    return 42

def get_y(x: int = Depends(get_x)) -> int:
    return x * 2

@app.get("/")
def root(y: int = Depends(get_y)):
    return {"y": y}   # → {"y": 84}
```

FastAPI calls `get_x()`, passes the result to `get_y()`, then passes that result to `root()`. The route function never calls either dependency directly.

By default, a dependency is called **once per request** even if it appears in multiple places. This is why `get_db` is safe to list in multiple dependencies — the same session object is reused.

```
# Both dependencies below receive the same Session object:
def get_food(food_id: int, db: Session = Depends(get_db)):  ...
def get_user(db: Session = Depends(get_db)):                ...
```

To disable caching (call the dependency fresh every time it appears), use `Depends(fn, use_cache=False)`. This is rarely needed.

Every `GET /foods` call will eventually need `skip` and `limit` query parameters. Rather than repeating them across multiple routes, extract them into a reusable callable.

```
# app/dependencies.py
from fastapi import Query

class Pagination:
    """Reusable skip/limit pagination parameters."""

    def __init__(
        self,
        skip:  int = Query(default=0,  ge=0,         description="Records to skip"),
        limit: int = Query(default=10, ge=1, le=100, description="Max records to return"),
    ):
        self.skip  = skip
        self.limit = limit
```

Using a class instead of a function makes the object's attributes explicit (`page.skip`, `page.limit`) and lets you add methods later if needed (e.g., `page.offset_limit()` for SQL).

In the router:

```
from app.dependencies import Pagination

@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category:        Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]         = Query(default=None),
    page:            Pagination             = Depends(Pagination),
    db:              Session                = Depends(get_db),
):
    return crud.get_foods(
        db,
        category=category,
        expiring_before=expiring_before,
        skip=page.skip,
        limit=page.limit,
    )
```

And in `crud.get_foods`, add the `skip` parameter:

```
def get_foods(
    db: Session,
```

```
    *,
    category:        Optional[CategoryEnum] = None,
    expiring_before: Optional[date]         = None,
    skip:            int                    = 0,
    limit:           int                    = 10,
) -> list[FoodItemModel]:
    query = db.query(FoodItemModel)
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    if expiring_before:
        query = query.filter(FoodItemModel.expiry_date <= expiring_before)
    return query.offset(skip).limit(limit).all()
```

Every route that touches a single food item (`GET /foods/{id}`, `PUT`, `PATCH`, `DELETE`) currently repeats the same pattern: call `crud.get_food()`, check `None`, raise 404. Move that logic into a dependency:

```
# app/dependencies.py (continued)
from fastapi import Depends, HTTPException, Path
from sqlalchemy.orm import Session
from app import crud
from app.database import get_db
from app.models import FoodItem as FoodItemModel

def get_valid_food(
    food_id: int     = Path(..., ge=1),
    db:      Session = Depends(get_db),
) -> FoodItemModel:
    """Resolve {food_id} to a FoodItem, or raise 404."""
    food = crud.get_food(db, food_id)
    if food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return food
```

Now the four routes that previously had identical lookup code become:

```
@router.get("/{food_id}", response_model=FoodItemResponse)
def get_food(food: FoodItemModel = Depends(get_valid_food)):
    return food


@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(
    food_in: FoodItemCreate,
    food:    FoodItemModel = Depends(get_valid_food),
    db:      Session       = Depends(get_db),
):
    return crud.replace_food(db, food.id, food_in)


@router.patch("/{food_id}", response_model=FoodItemResponse)
def update_food(
    updates: FoodItemUpdate,
    food:    FoodItemModel = Depends(get_valid_food),
    db:      Session       = Depends(get_db),
):
    return crud.update_food(db, food.id, updates)
```

```python
@router.delete("/{food_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_food(
    food: FoodItemModel = Depends(get_valid_food),
    db:   Session       = Depends(get_db),
):
    crud.delete_food(db, food.id)
```

Notice that the 404 check is gone from every handler — it now lives in exactly one place.

> **Note on session sharing:** `get_valid_food` uses `Depends(get_db)` internally, and the route also uses `Depends(get_db)`. Because of dependency caching, both receive **the same `Session` object**. The food object loaded by `get_valid_food` is already in that session's identity map, so the subsequent `crud.update_food` call finds it without an extra query.

Authentication is covered in Lesson 15. But the dependency pattern for "who is logged in" is worth introducing now, because routes that need it should declare it from the start — retrofitting authentication later is messier than having a stub in place.

```python
# app/dependencies.py (continued)
from dataclasses import dataclass

@dataclass
class CurrentUser:
    id:       int
    username: str
    is_admin: bool = False

def get_current_user() -> CurrentUser:
    """
    Stub — always returns a hardcoded user.
    Lesson 15 replaces this with real JWT validation.
    """
    return CurrentUser(id=1, username="philip", is_admin=True)
```

Use it in a route:

```python
@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(
    food:         FoodItemCreate = ...,
    db:           Session        = Depends(get_db),
    current_user: CurrentUser    = Depends(get_current_user),
):
    print(f"Created by {current_user.username}")
    return crud.create_food(db, food)
```

When Lesson 15 replaces `get_current_user` with a real JWT implementation, every route that declares `Depends(get_current_user)` gains real authentication with no other changes.

A dependency doesn't have to return a useful value — it can exist purely to assert a precondition. Here is an admin guard that raises 403 if the user is not an admin:

```
# app/dependencies.py (continued)
def require_admin(current_user: CurrentUser = Depends(get_current_user)) -> None:
    if not current_user.is_admin:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Admin access required",
        )
```

Add it to a route with `dependencies=` on the decorator — no parameter needed in the handler:

```
@router.delete(
    "/{food_id}",
    status_code=status.HTTP_204_NO_CONTENT,
    dependencies=[Depends(require_admin)],
)
def delete_food(
    food: FoodItemModel = Depends(get_valid_food),
    db:   Session       = Depends(get_db),
):
    crud.delete_food(db, food.id)
```

Or apply it to every route in the router at once:

```
router = APIRouter(
    prefix="/foods",
    tags=["foods"],
    dependencies=[Depends(require_admin)],
)
```

Dependencies can depend on other dependencies to any depth FastAPI resolves the graph automatically:

```
Request
  ■■ delete_food()
       ■■ get_valid_food()     ← depends on get_db, path param
       ■    ■■ get_db()        ← yields session (cached)
       ■■ get_db()             ← same session (cache hit)
       ■■ require_admin()      ← depends on get_current_user
            ■■ get_current_user() ← returns CurrentUser
```

The session is created once; `get_current_user` is called once. FastAPI builds this graph from type annotations alone.

Dependencies that use `yield` run cleanup code after the response is sent, just like `get_db`:

```
def get_db():
    db = SessionLocal()
    try:
        yield db          # ← route body runs here
    finally:
        db.close()        # ← always runs, even on exception
```

You can write your own generator dependencies the same way:

```
import time

def log_request_time():
    start = time.perf_counter()
    yield                              # route runs
    elapsed = time.perf_counter() - start
    print(f"Request took {elapsed:.3f}s")
```

Apply it as a router-level dependency to time every route:

```
router = APIRouter(
    prefix="/foods",
    dependencies=[Depends(log_request_time)],
)
```

```
# app/dependencies.py
from dataclasses import dataclass
from fastapi import Depends, HTTPException, Path, Query, status
from sqlalchemy.orm import Session
from app import crud
from app.database import get_db
from app.models import FoodItem as FoodItemModel


class Pagination:
    def __init__(
        self,
        skip:  int = Query(default=0,  ge=0,         description="Records to skip"),
        limit: int = Query(default=10, ge=1, le=100, description="Max records to return"),
    ):
        self.skip  = skip
        self.limit = limit


@dataclass
class CurrentUser:
    id:       int
    username: str
    is_admin: bool = False


def get_current_user() -> CurrentUser:
    """Stub. Replace with JWT validation in Lesson 15."""
    return CurrentUser(id=1, username="philip", is_admin=True)
```

```
def get_valid_food(
    food_id: int     = Path(..., ge=1),
    db:      Session = Depends(get_db),
) -> FoodItemModel:
    food = crud.get_food(db, food_id)
    if food is None:
        raise HTTPException(status_code=404, detail="Food item not found")
    return food


def require_admin(current_user: CurrentUser = Depends(get_current_user)) -> None:
    if not current_user.is_admin:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Admin access required",
        )
```

```
# app/routers/foods.py
from datetime import date
from typing import Optional

from fastapi import APIRouter, Depends, Query, status
from sqlalchemy.orm import Session

from app import crud
from app.database import get_db
from app.dependencies import CurrentUser, FoodItemModel, Pagination, get_current_user, get_valid_food, require_admin
from app.schemas import CategoryEnum, FoodItemCreate, FoodItemResponse, FoodItemUpdate

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


@router.get("", response_model=list[FoodItemResponse])
def get_foods(
    category:        Optional[CategoryEnum] = Query(default=None),
    expiring_before: Optional[date]         = Query(default=None),
    page:            Pagination             = Depends(Pagination),
    db:              Session                = Depends(get_db),
):
    return crud.get_foods(
        db,
        category=category,
        expiring_before=expiring_before,
        skip=page.skip,
        limit=page.limit,
    )


@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(
    food:         FoodItemCreate,
    db:           Session     = Depends(get_db),
    current_user: CurrentUser = Depends(get_current_user),
):
    return crud.create_food(db, food)


@router.get("/expiring-soon", response_model=list[FoodItemResponse])
```

```
def get_expiring_soon(
    days: int     = Query(default=7, ge=1, le=30),
    db:   Session = Depends(get_db),
):
    return crud.get_expiring_soon(db, days=days)


@router.get("/{food_id}", response_model=FoodItemResponse)
def get_food(food: FoodItemModel = Depends(get_valid_food)):
    return food


@router.put("/{food_id}", response_model=FoodItemResponse)
def replace_food(
    food_in: FoodItemCreate,
    food:    FoodItemModel = Depends(get_valid_food),
    db:      Session       = Depends(get_db),
):
    return crud.replace_food(db, food.id, food_in)
    # ... (truncated for readability)
```

```
food_tracker/
■■■ alembic/
■■■ alembic.ini
■■■ app/
■   ■■■ __init__.py
■   ■■■ main.py
■   ■■■ config.py
■   ■■■ database.py
■   ■■■ models.py
■   ■■■ schemas.py
■   ■■■ crud.py
■   ■■■ dependencies.py   ← NEW: Pagination, CurrentUser, get_valid_food, require_admin
■   ■■■ routers/
■       ■■■ __init__.py
■       ■■■ foods.py      ← updated: uses dependencies, no more inline 404 checks
■■■ food_tracker.db
■■■ requirements.txt
```

---

1. The `get_expiring_soon` endpoint doesn't use `Pagination` yet. Add it — accept `skip` and `limit` via the `Pagination` dependency and pass them through to a new `crud.get_expiring_soon` signature that uses `.offset(skip).limit(limit)`.

2. Create an `ExpirySoonQuery` dependency class (similar to `Pagination`) that bundles both the `days` parameter and `Pagination` into a single object:

```
class ExpirySoonQuery:
    def __init__(
        self,
```

```
        days: int        = Query(default=7, ge=1, le=30),
        page: Pagination = Depends(Pagination),
    ):
        self.days = days
        self.page = page
```

Use it in `get_expiring_soon`. Notice that you can nest dependencies inside `__init__` with `Depends()`.

3. Add a `require_admin` guard to the `POST /foods` route (use the `dependencies=` parameter on the decorator) and verify in `/docs` that it appears in the endpoint description. Then change `get_current_user` to return `is_admin=False` and confirm you get a 403.

- `Depends()` **accepts any callable** — functions, classes (called as constructors), async functions, and generators.
- **Dependencies are cached per request** — the same `db` session is reused across all dependencies in a single request without extra queries.
- **Generator dependencies run cleanup after the response** — `get_db`'s `finally` block closes the session even if the route raises an exception.
- `get_valid_food` **eliminates the repeated 404 pattern** — one place to change if the lookup logic ever needs adjusting.
- `dependencies=[]` **on a decorator or router applies a guard without adding a parameter to the handler** — ideal for auth and admin checks.
- **Stub dependencies are worth adding early** — `get_current_user` can return a hardcoded user today and be replaced with JWT validation in Lesson 15 with no route changes.

Lesson 12 digs into filtering and sorting — adding query parameters that produce more complex SQLAlchemy queries, handling date ranges, sorting by column, and returning metadata like total counts alongside paginated results.

LESSON 12

FastAPI Lesson 12 — Filtering & Sorting

The current `GET /foods` endpoint accepts a category filter and a single date ceiling. Real APIs need more: date ranges, multi-field sorting, and response envelopes that tell the client how many total records exist so it can render pagination controls. This lesson adds all three while keeping query logic inside `crud.py` and route signatures clean through a dependency.

The existing `expiring_before` filter selects items that expire on or before a given date. Adding `expiring_after` turns it into a proper date range.

Update `crud.get_foods`:

```
# app/crud.py
from datetime import date
from typing import Optional
from sqlalchemy.orm import Session
from app.models import FoodItem as FoodItemModel
from app.schemas import CategoryEnum

def get_foods(
    db: Session,
    *,
    category:        Optional[CategoryEnum] = None,
    expiring_after:  Optional[date]         = None,
    expiring_before: Optional[date]         = None,
    sort_by:         str                    = "expiry_date",
    sort_dir:        str                    = "asc",
    skip:            int                    = 0,
    limit:           int                    = 10,
) -> tuple[list[FoodItemModel], int]:
    """
    Returns (items, total_count).
    total_count is the count before pagination — used by the response envelope.
    """
    query = db.query(FoodItemModel)

    # ── Filters ──────────────────────────────────────────────────
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    if expiring_after:
        query = query.filter(FoodItemModel.expiry_date >= expiring_after)
    if expiring_before:
        query = query.filter(FoodItemModel.expiry_date <= expiring_before)
```

```
    # ■■ Count before pagination ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    total = query.count()

    # ■■ Sorting ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    sort_column = getattr(FoodItemModel, sort_by, FoodItemModel.expiry_date)
    if sort_dir == "desc":
        query = query.order_by(sort_column.desc())
    else:
        query = query.order_by(sort_column.asc())

    # ■■ Pagination ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    items = query.offset(skip).limit(limit).all()

    return items, total
```

Two things changed:

- The function now accepts `expiring_after`, `sort_by`, `sort_dir`, `skip`, and `limit`.
- It returns a `tuple[list, int]` instead of just a list — the integer is the pre-pagination total count.

`sort_by` names the column; `sort_dir` names the direction. Using `getattr` on the model is safe here because we validate the column name in the dependency (next section) before it ever reaches `crud.py`.

Allowed column names for the food tracker:

sort_by value	Maps to
`expiry_date`	expiry date column
`name`	item name
`category`	category enum
`created_at`	insert timestamp

Rather than adding five new parameters to the route signature, bundle them into a dependency. This follows the same pattern as `Pagination` from Lesson 11.

```
# app/dependencies.py (additions)
from datetime import date
from typing import Literal, Optional
from fastapi import Depends, Query
from app.schemas import CategoryEnum
from app.dependencies import Pagination

SORTABLE_COLUMNS = {"expiry_date", "name", "category", "created_at"}

class FoodFilterQuery:
```

```
    """
    Bundles all GET /foods filter and sort parameters.
    Validates sort_by against the allowed column set.
    """
    def __init__(
        self,
        category:        Optional[CategoryEnum] = Query(default=None),
        expiring_after:  Optional[date]         = Query(default=None, description="Items expiring on or after this da
        expiring_before: Optional[date]         = Query(default=None, description="Items expiring on or before this d
        sort_by:         str                    = Query(default="expiry_date", description=f"Column to sort by: {SORT
        sort_dir:        Literal["asc", "desc"] = Query(default="asc"),
        page:            Pagination             = Depends(Pagination),
    ):
        if sort_by not in SORTABLE_COLUMNS:
            raise HTTPException(
                status_code=422,
                detail=f"sort_by must be one of {sorted(SORTABLE_COLUMNS)}",
            )
        self.category        = category
        self.expiring_after  = expiring_after
        self.expiring_before = expiring_before
        self.sort_by         = sort_by
        self.sort_dir        = sort_dir
        self.page            = page
```

Using `Literal["asc", "desc"]` for `sort_dir` means FastAPI validates and documents the allowed values automatically — no manual check needed.

A bare list tells the client nothing about total record count. A response envelope wraps the list with metadata the client needs for pagination controls:

```
# app/schemas.py (addition)
from pydantic import BaseModel

class FoodListResponse(BaseModel):
    items:   list[FoodItemResponse]
    total:   int
    skip:    int
    limit:   int
    has_more: bool

    model_config = ConfigDict(from_attributes=True)
```

`has_more` is a computed convenience: `skip + len(items) < total`. The client uses it to decide whether to show a "next page" button without doing arithmetic itself.

```
# app/routers/foods.py — updated GET /foods
from app.dependencies import FoodFilterQuery
from app.schemas import FoodListResponse
```

```python
@router.get("", response_model=FoodListResponse)
def get_foods(
    filters: FoodFilterQuery = Depends(FoodFilterQuery),
    db:      Session         = Depends(get_db),
):
    items, total = crud.get_foods(
        db,
        category        = filters.category,
        expiring_after  = filters.expiring_after,
        expiring_before = filters.expiring_before,
        sort_by         = filters.sort_by,
        sort_dir        = filters.sort_dir,
        skip            = filters.page.skip,
        limit           = filters.page.limit,
    )
    return FoodListResponse(
        items    = items,
        total    = total,
        skip     = filters.page.skip,
        limit    = filters.page.limit,
        has_more = filters.page.skip + len(items) < total,
    )
```

The route handler is still short. All validation is in the dependency; all query logic is in `crud.py`.

```bash
# All dairy items, sorted newest-expiring first
GET /foods?category=Dairy&sort_by=expiry_date&sort_dir=desc

# Items expiring this week, page 2
GET /foods?expiring_before=2026-06-14&skip=10&limit=10

# Items expiring in a specific range, sorted by name
GET /foods?expiring_after=2026-06-01&expiring_before=2026-06-30&sort_by=name

# Invalid sort column — returns 422 immediately (rejected in dependency)
GET /foods?sort_by=invalid_column
```

Example response for the second request:

```json
{
  "items": [ ... ],
  "total": 23,
  "skip":  10,
  "limit": 10,
  "has_more": true
}
```

`get_expiring_soon` is a specialised query that still works well as its own endpoint. Update it to also support pagination:

```
# app/crud.py
def get_expiring_soon(
    db: Session,
    *,
    days:  int = 7,
    skip:  int = 0,
    limit: int = 10,
) -> tuple[list[FoodItemModel], int]:
    from datetime import timedelta
    cutoff = date.today() + timedelta(days=days)
    query  = db.query(FoodItemModel).filter(FoodItemModel.expiry_date <= cutoff)
    total  = query.count()
    items  = query.order_by(FoodItemModel.expiry_date.asc()).offset(skip).limit(limit).all()
    return items, total
```

And the updated route:

```
@router.get("/expiring-soon", response_model=FoodListResponse)
def get_expiring_soon(
    days: int        = Query(default=7, ge=1, le=30),
    page: Pagination = Depends(Pagination),
    db:   Session    = Depends(get_db),
):
    items, total = crud.get_expiring_soon(db, days=days, skip=page.skip, limit=page.limit)
    return FoodListResponse(
        items    = items,
        total    = total,
        skip     = page.skip,
        limit    = page.limit,
        has_more = page.skip + len(items) < total,
    )
```

Each filter is applied conditionally by reassigning `query`. SQLAlchemy builds the SQL lazily — nothing is sent to the database until `.count()` or `.all()` is called:

```
query = db.query(FoodItemModel)              # SELECT * FROM food_items
if category:
    query = query.filter(...)                 # ... WHERE category = ?
if expiring_after:
    query = query.filter(...)                 # ... AND expiry_date >= ?
total = query.count()                         # ← first DB hit: SELECT COUNT(*)
items = query.offset(skip).limit(limit).all() # ← second DB hit: SELECT * LIMIT ...
```

The two DB hits (count, then items) are intentional. The count uses the same filter conditions as the items query, so the total is always consistent with the page you're on.

```
# app/crud.py
from datetime import date, timedelta
```

```
from typing import Optional
from sqlalchemy.orm import Session
from app.models import FoodItem as FoodItemModel
from app.schemas import CategoryEnum, FoodItemCreate, FoodItemUpdate

SORTABLE_COLUMNS = {"expiry_date", "name", "category", "created_at"}


def get_food(db: Session, food_id: int) -> FoodItemModel | None:
    return db.query(FoodItemModel).filter(FoodItemModel.id == food_id).first()


def get_foods(
    db: Session,
    *,
    category:        Optional[CategoryEnum] = None,
    expiring_after:  Optional[date]         = None,
    expiring_before: Optional[date]         = None,
    sort_by:         str                    = "expiry_date",
    sort_dir:        str                    = "asc",
    skip:            int                    = 0,
    limit:           int                    = 10,
) -> tuple[list[FoodItemModel], int]:
    query = db.query(FoodItemModel)
    if category:
        query = query.filter(FoodItemModel.category == category.value)
    if expiring_after:
        query = query.filter(FoodItemModel.expiry_date >= expiring_after)
    if expiring_before:
        query = query.filter(FoodItemModel.expiry_date <= expiring_before)
    total = query.count()
    col = getattr(FoodItemModel, sort_by, FoodItemModel.expiry_date)
    query = query.order_by(col.desc() if sort_dir == "desc" else col.asc())
    return query.offset(skip).limit(limit).all(), total


def get_expiring_soon(
    db: Session,
    *,
    days:  int = 7,
    skip:  int = 0,
    limit: int = 10,
) -> tuple[list[FoodItemModel], int]:
    cutoff = date.today() + timedelta(days=days)
    query  = db.query(FoodItemModel).filter(FoodItemModel.expiry_date <= cutoff)
    total  = query.count()
    items  = query.order_by(FoodItemModel.expiry_date.asc()).offset(skip).limit(limit).all()
    return items, total


def create_food(db: Session, food: FoodItemCreate) -> FoodItemModel:
    db_food = FoodItemModel(**food.model_dump())
    db.add(db_food)
    db.commit()
    db.refresh(db_food)
    return db_food


    # ... (truncated for readability)
```

1. Add a `name_contains` filter to `FoodFilterQuery` and `crud.get_foods` that does a case-insensitive partial match on the item name:

```
# In FoodFilterQuery.__init__:
name_contains: Optional[str] = Query(default=None, description="Filter by name (case-insensitive substring)")

# In crud.get_foods:
if name_contains:
    query = query.filter(FoodItemModel.name.ilike(f"%{name_contains}%"))
```

Test it with `GET /foods?name_contains=milk`.

2. The `has_more` boolean in `FoodListResponse` is currently computed in the route handler. Move it to a `@property` or `@computed_field` on the schema so the route just returns the raw values and Pydantic computes the rest.

```
from pydantic import computed_field

class FoodListResponse(BaseModel):
    items:  list[FoodItemResponse]
    total:  int
    skip:   int
    limit:  int

    @computed_field
    @property
    def has_more(self) -> bool:
        return self.skip + len(self.items) < self.total
```

3. Add a second sort column — items with the same primary sort value should be ordered by `name` as a tiebreaker. Update `crud.get_foods` to use `query.order_by(primary_col, FoodItemModel.name)`.

- **Return `(items, total)` from list crud functions** — the count query runs against the same filters as the item query, so the total is always consistent with the current page.
- **Validate `sort_by` in the dependency, not in `crud.py`** — the dependency is HTTP-aware and can raise a 422; the crud function is not.
- `Literal["asc", "desc"]` gives FastAPI automatic validation and documentation of the allowed values at no extra cost.
- **SQLAlchemy queries are lazy** — filters are accumulated on the `query` object before any SQL is sent. `.count()` and `.all()` are the only two database round-trips.
- **The response envelope (`FoodListResponse`)** gives clients the data they need to render pagination controls without doing server-side arithmetic themselves.

Lesson 13 covers background tasks — using FastAPI's `BackgroundTasks` to send notifications after a response is returned, and looking at when to reach for a task queue instead.

LESSON 13

FastAPI Lesson 13 — Background Tasks

Some work shouldn't block the HTTP response. Sending an email, writing an audit log, or triggering a downstream notification can all happen *after* the client has received its `201 Created`. FastAPI's built-in `BackgroundTasks` handles exactly this — lightweight post-response work that runs in the same process. This lesson covers how it works, where it fits in the food tracker, and when it is *not* the right tool.

FastAPI provides a `BackgroundTasks` object as a dependency you declare in the route signature. You add callables to it with `.add_task()`. After the response is sent to the client, FastAPI runs each task in the order they were added.

```
from fastapi import BackgroundTasks

def write_log(message: str) -> None:
    with open("log.txt", "a") as f:
        f.write(message + "\n")

@app.post("/items")
def create_item(background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, "item created")
    return {"status": "created"}
```

The client gets `{"status": "created"}` immediately. `write_log` runs after the response is sent, in the same thread as the server.

Key behaviours

- Tasks run **in the same process and thread** as the web server — they share memory with the running application.
- They run **after the response body is fully sent**, not concurrently with sending it.
- If a task raises an exception, the exception is logged but does not change the HTTP response the client already received.
- `BackgroundTasks` can be injected into route handlers **and** into dependencies — useful when a dependency needs to schedule cleanup.

When a food item is created with an expiry date less than seven days away, notify the user immediately after the `201` is returned.

```
# app/notifications.py
import logging
from datetime import date

logger = logging.getLogger(__name__)

def notify_expiring_soon(item_name: str, expiry_date: date) -> None:
    """
    Placeholder notification. In production, replace the log statement
    with an email, push notification, or Slack message.
    """
    days_left = (expiry_date - date.today()).days
    logger.warning(
        "EXPIRY ALERT: '%s' expires in %d day(s) on %s.",
        item_name,
        days_left,
        expiry_date,
    )


# app/routers/foods.py — updated POST
from fastapi import BackgroundTasks
from app.notifications import notify_expiring_soon
from datetime import date

WARNING_THRESHOLD_DAYS = 7

@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(
    food:             FoodItemCreate,
    background_tasks: BackgroundTasks,
    db:               Session     = Depends(get_db),
    current_user:     CurrentUser = Depends(get_current_user),
):
    db_food = crud.create_food(db, food)

    days_until_expiry = (db_food.expiry_date - date.today()).days
    if days_until_expiry <= WARNING_THRESHOLD_DAYS:
        background_tasks.add_task(
            notify_expiring_soon,
            db_food.name,
            db_food.expiry_date,
        )

    return db_food
```

The route adds the notification task only when warranted, then returns immediately. The notification function runs after the `201` is already on its way to the client.

Every deletion should write an audit record. The client doesn't need to wait for the write to succeed.

```
# app/notifications.py (continued)
def log_deletion(item_id: int, item_name: str, deleted_by: str) -> None:
    logger.info(
        "DELETED: food_id=%d name='%s' by user='%s'",
        item_id,
        item_name,
        deleted_by,
    )


# app/routers/foods.py — updated DELETE
@router.delete(
    "/{food_id}",
    status_code=status.HTTP_204_NO_CONTENT,
    dependencies=[Depends(require_admin)],
)
def delete_food(
    background_tasks: BackgroundTasks,
    food:             FoodItemModel = Depends(get_valid_food),
    db:               Session       = Depends(get_db),
    current_user:     CurrentUser   = Depends(get_current_user),
):
    # Capture name before deletion — after commit the ORM object may be expired
    item_id   = food.id
    item_name = food.name

    crud.delete_food(db, food.id)

    background_tasks.add_task(log_deletion, item_id, item_name, current_user.username)
```

> **Important:** capture any values you need from the ORM object *before* the database commit. After `crud.delete_food` calls `db.commit()`, SQLAlchemy may expire the object's attributes. If `log_deletion` tried to access `food.name` later, it could trigger a lazy load on a closed or committed session.

`BackgroundTasks` can be declared inside a dependency, not just in the route:

```
# app/dependencies.py
import time

def log_request_time(background_tasks: BackgroundTasks) -> None:
    start = time.perf_counter()

    def _log():
        elapsed = time.perf_counter() - start
        logger.info("Request completed in %.3fs", elapsed)

    background_tasks.add_task(_log)
```

Apply it as a router-level dependency:

```
router = APIRouter(
    prefix="/foods",
    tags=["foods"],
    dependencies=[Depends(log_request_time)],
)
```

FastAPI passes the same `BackgroundTasks` instance to the dependency that it passes to the route — tasks added anywhere in the dependency graph all run after the response.

Background tasks can use a database session, but they must manage the session lifecycle themselves — the request-scoped `get_db` session is closed before the task runs.

```
# app/notifications.py
from app.database import SessionLocal
from app import crud

def persist_audit_log(item_id: int, action: str, user: str) -> None:
    """Write an audit record to the database after the response is sent."""
    db = SessionLocal()
    try:
        # Example: insert into an audit_logs table via a crud function
        # crud.create_audit_log(db, item_id=item_id, action=action, user=user)
        logger.info("Audit: %s on food_id=%d by %s", action, item_id, user)
    finally:
        db.close()
```

Never pass the route's `db` session to a background task. That session is already closed (the `get_db` generator's `finally` block ran when the response was prepared).

Background task functions can be either regular (`def`) or async (`async def`). FastAPI runs sync tasks in a thread pool executor and awaits async tasks. For the food tracker's use cases (logging, email, simple DB writes), plain `def` is fine and easier to reason about.

```
# Both work:
background_tasks.add_task(sync_function, arg1, arg2)
background_tasks.add_task(async_function, arg1, arg2)
```

`BackgroundTasks` is built for **lightweight, fast, non-critical work**. Know its limits:

Scenario	BackgroundTasks?	Better option
Send a simple log entry	✅ Yes	—
Send a single email	✅ Yes	—
Process an uploaded image	❌ No	Celery / ARQ / RQ

Scenario	BackgroundTasks?	Better option
Retry failed work automatically	⬛ No	Task queue with retry logic
Distribute work across workers	⬛ No	Celery + Redis/RabbitMQ
Work that must survive a server crash	⬛ No	Persistent task queue
Report task progress to client	⬛ No	WebSocket or SSE + task queue

`BackgroundTasks` tasks are lost if the server process dies while they are queued. For critical work — payment processing, data exports, third-party API calls with retry logic — use a dedicated task queue (Celery, ARQ, or RQ) backed by Redis.

```python
# app/notifications.py
import logging
from datetime import date

from app.database import SessionLocal

logger = logging.getLogger(__name__)


def notify_expiring_soon(item_name: str, expiry_date: date) -> None:
    """Alert that a food item is expiring soon."""
    days_left = (expiry_date - date.today()).days
    logger.warning(
        "EXPIRY ALERT: '%s' expires in %d day(s) on %s.",
        item_name,
        days_left,
        expiry_date,
    )


def log_deletion(item_id: int, item_name: str, deleted_by: str) -> None:
    """Audit log for food item deletion."""
    logger.info(
        "DELETED: food_id=%d name='%s' by user='%s'",
        item_id,
        item_name,
        deleted_by,
    )


def persist_audit_log(item_id: int, action: str, user: str) -> None:
    """Write an audit record to the database (manages its own session)."""
    db = SessionLocal()
    try:
        logger.info("Audit: %s on food_id=%d by %s", action, item_id, user)
        # crud.create_audit_log(db, ...)  ← uncomment when audit table exists
    finally:
        db.close()
```

```
food_tracker/
├── alembic/
├── alembic.ini
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── config.py
│   ├── database.py
│   ├── models.py
│   ├── schemas.py
│   ├── crud.py
│   ├── dependencies.py
│   ├── notifications.py   ← NEW: background task functions
│   └── routers/
│       ├── __init__.py
│       └── foods.py       ← updated: BackgroundTasks on POST and DELETE
├── food_tracker.db
└── requirements.txt
```

---

1. Add a background task to `PATCH /foods/{food_id}` that logs which fields were changed and their new values:

```
def log_update(item_id: int, changes: dict, user: str) -> None:
    logger.info("UPDATED food_id=%d fields=%s by user='%s'", item_id, changes, user)
```

Pass `updates.model_dump(exclude_unset=True)` as the `changes` argument.

2. Add a `notify_expiring_soon` check to `PUT /foods/{food_id}` (full replacement). If the new expiry date is within the warning threshold, schedule the notification background task.

3. Test that the response arrives before the task runs. Add a `time.sleep(2)` to `notify_expiring_soon`, POST a near-expiry item, and observe that the `201` response arrives immediately while the sleep happens in the background. Then remove the sleep.

- `BackgroundTasks` **runs work after the response is sent** — the client never waits for the task to complete.
- **Tasks share the process** — they have access to the same in-memory state, but not to the request-scoped database session (which is already closed).
- **Capture ORM values before commit** — after `db.commit()`, SQLAlchemy may expire attribute access on ORM objects.
- **Background tasks that need the database must open their own session** via `SessionLocal()` and close it in a `finally` block.

- **Tasks can be added from dependencies**, not just route handlers — FastAPI passes the same `BackgroundTasks` instance throughout the dependency graph.
- **Use a proper task queue** (Celery, ARQ) for retryable, distributed, or crash-safe work.

Lesson 14 covers error handling — custom exception handlers, consistent error response shapes, validation error customisation, and how to avoid leaking internal details in error messages.

LESSON 14

FastAPI Lesson 14 — Error Handling

FastAPI's default error responses are functional but inconsistent — a 422 from Pydantic looks completely different from a 404 raised by `HTTPException`. Production APIs should return a uniform error envelope on every failure, hide internal details from clients, and give developers enough context to debug problems. This lesson adds a consistent error shape, custom exception handlers, a domain-specific exception class, and safe validation error formatting.

Out of the box, FastAPI produces two distinct error shapes:

HTTPException (404, 403, etc.):

```
{ "detail": "Food item not found" }
```

RequestValidationError (422 — bad request body or query params):

```
{
  "detail": [
    {
      "type": "missing",
      "loc": ["body", "name"],
      "msg": "Field required",
      "input": {},
      "url": "https://errors.pydantic.dev/..."
    }
  ]
}
```

Two problems: the shapes differ, and the 422 body can expose internal field names, nested model paths, and Pydantic URLs — more than a client needs, and potentially more than you want public.

Define the error shape once. Every error response from the API will use it:

```
# app/errors.py
from pydantic import BaseModel
```

```
class ErrorDetail(BaseModel):
    field:   str | None = None   # which field caused the problem, if any
    message: str                 # human-readable description

class ErrorResponse(BaseModel):
    status:  int               # HTTP status code (echoed for convenience)
    error:   str               # short machine-readable code, e.g. "not_found"
    details: list[ErrorDetail] # one entry per problem
```

Example output for a 404:

```
{
  "status":  404,
  "error":   "not_found",
  "details": [{ "field": null, "message": "Food item not found" }]
}
```

Example output for a 422:

```
{
  "status":  422,
  "error":   "validation_error",
  "details": [
    { "field": "name",        "message": "Field required" },
    { "field": "expiry_date", "message": "Value is not a valid date" }
  ]
}
```

Register a handler on the FastAPI app that intercepts every `HTTPException` and reformats it:

```
# app/errors.py (continued)
from fastapi import FastAPI, HTTPException, Request
from fastapi.responses import JSONResponse

def _http_error_code(status_code: int) -> str:
    """Map HTTP status codes to short machine-readable error codes."""
    return {
        400: "bad_request",
        401: "unauthorised",
        403: "forbidden",
        404: "not_found",
        409: "conflict",
        422: "validation_error",
        500: "internal_error",
    }.get(status_code, "error")


async def http_exception_handler(request: Request, exc: HTTPException) -> JSONResponse:
    return JSONResponse(
        status_code=exc.status_code,
        content=ErrorResponse(
            status  = exc.status_code,
            error   = _http_error_code(exc.status_code),
            details = [ErrorDetail(message=str(exc.detail))],
        ).model_dump(),
    )
```

```
# app/main.py
from fastapi import FastAPI, HTTPException
from fastapi.exceptions import RequestValidationError
from app.errors import http_exception_handler, validation_exception_handler

app = FastAPI()
app.add_exception_handler(HTTPException, http_exception_handler)
app.add_exception_handler(RequestValidationError, validation_exception_handler)
```

The `RequestValidationError` handler extracts only the field name and message — no Pydantic internals:

```
# app/errors.py (continued)
from fastapi.exceptions import RequestValidationError

async def validation_exception_handler(
    request: Request, exc: RequestValidationError
) -> JSONResponse:
    details = []
    for error in exc.errors():
        # loc is a tuple like ("body", "name") or ("query", "limit")
        # We take the last element as the field name
        loc    = error.get("loc", [])
        field  = str(loc[-1]) if loc else None
        msg    = error.get("msg", "Invalid value")
        details.append(ErrorDetail(field=field, message=msg))

    return JSONResponse(
        status_code=422,
        content=ErrorResponse(
            status  = 422,
            error   = "validation_error",
            details = details,
        ).model_dump(),
    )
```

What the client now gets instead of the full Pydantic error dump:

```
{
  "status":  422,
  "error":   "validation_error",
  "details": [
    { "field": "expiry_date", "message": "Value is not a valid date" }
  ]
}
```

No Pydantic URLs. No internal path prefixes beyond the field name. No `input` echo.

Rather than scattering raw `HTTPException` calls throughout the codebase, define typed exceptions that encode their own status code and error code. This makes intent explicit and keeps HTTP details out of business

logic:

```
# app/errors.py (continued)
from fastapi import status as http_status

class AppError(Exception):
    """Base class for all application-level errors."""
    status_code: int = 500
    error_code:  str = "internal_error"
    message:     str = "An unexpected error occurred."

    def __init__(self, message: str | None = None):
        self.message = message or self.__class__.message
        super().__init__(self.message)


class NotFoundError(AppError):
    status_code = 404
    error_code  = "not_found"
    message     = "Resource not found."


class ConflictError(AppError):
    status_code = 409
    error_code  = "conflict"
    message     = "Resource already exists."


class ForbiddenError(AppError):
    status_code = 403
    error_code  = "forbidden"
    message     = "You do not have permission to perform this action."
```

Register a handler for `AppError`:

```
# app/errors.py (continued)
async def app_error_handler(request: Request, exc: AppError) -> JSONResponse:
    return JSONResponse(
        status_code=exc.status_code,
        content=ErrorResponse(
            status  = exc.status_code,
            error   = exc.error_code,
            details = [ErrorDetail(message=exc.message)],
        ).model_dump(),
    )


# app/main.py — add the third handler
app.add_exception_handler(AppError, app_error_handler)
```

Now instead of:

```
raise HTTPException(status_code=404, detail="Food item not found")
```

You write:

```
raise NotFoundError("Food item not found")
```

The error handler converts it to the same JSON envelope. The raise site reads like business logic, not HTTP plumbing.

Unhandled exceptions become 500 errors. By default, FastAPI returns the exception message, which can include database schema names, file paths, or stack details. Override it:

```
# app/errors.py (continued)
async def unhandled_exception_handler(request: Request, exc: Exception) -> JSONResponse:
    # Log the real exception with full traceback for developers
    import logging, traceback
    logging.getLogger(__name__).error(
        "Unhandled exception on %s %s:\n%s",
        request.method,
        request.url.path,
        traceback.format_exc(),
    )
    # Return a safe generic message to the client
    return JSONResponse(
        status_code=500,
        content=ErrorResponse(
            status  = 500,
            error   = "internal_error",
            details = [ErrorDetail(message="An unexpected error occurred.")],
        ).model_dump(),
    )


# app/main.py
app.add_exception_handler(Exception, unhandled_exception_handler)
```

Clients never see stack traces or database error text. Developers see it in the logs.

With `NotFoundError` available, update the dependency:

```
# app/dependencies.py
from app.errors import NotFoundError

def get_valid_food(
    food_id: int     = Path(..., ge=1),
    db:      Session = Depends(get_db),
) -> FoodItemModel:
    food = crud.get_food(db, food_id)
    if food is None:
        raise NotFoundError(f"Food item {food_id} not found.")
    return food
```

The response is identical to before — `404` with the uniform JSON envelope — but the raise site is cleaner and the exception carries semantic meaning.

```python
# app/errors.py
import logging
import traceback
from fastapi import HTTPException, Request
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse
from pydantic import BaseModel


# ── Envelope schema ──────────────────────────────────────────────────────────

class ErrorDetail(BaseModel):
    field:   str | None = None
    message: str

class ErrorResponse(BaseModel):
    status:  int
    error:   str
    details: list[ErrorDetail]


# ── Domain exceptions ────────────────────────────────────────────────────────

class AppError(Exception):
    status_code: int = 500
    error_code:  str = "internal_error"
    message:     str = "An unexpected error occurred."

    def __init__(self, message: str | None = None):
        self.message = message or self.__class__.message
        super().__init__(self.message)

class NotFoundError(AppError):
    status_code = 404
    error_code  = "not_found"
    message     = "Resource not found."

class ConflictError(AppError):
    status_code = 409
    error_code  = "conflict"
    message     = "Resource already exists."

class ForbiddenError(AppError):
    status_code = 403
    error_code  = "forbidden"
    message     = "You do not have permission to perform this action."


# ── Handlers ─────────────────────────────────────────────────────────────────

def _http_error_code(status_code: int) -> str:
    return {
        400: "bad_request", 401: "unauthorised", 403: "forbidden",
        404: "not_found",   409: "conflict",     422: "validation_error",
        500: "internal_error",
    }.get(status_code, "error")


async def http_exception_handler(request: Request, exc: HTTPException) -> JSONResponse:
    return JSONResponse(
    # ... (truncated for readability)
```

```python
# app/main.py
from contextlib import asynccontextmanager
from fastapi import FastAPI, HTTPException
from fastapi.exceptions import RequestValidationError
from app.errors import (
    AppError,
    app_error_handler,
    http_exception_handler,
    unhandled_exception_handler,
    validation_exception_handler,
)
from app.database import Base, engine
from app.routers import foods

@asynccontextmanager
async def lifespan(app: FastAPI):
    Base.metadata.create_all(bind=engine)
    yield

app = FastAPI(lifespan=lifespan)

app.add_exception_handler(HTTPException,           http_exception_handler)
app.add_exception_handler(RequestValidationError,  validation_exception_handler)
app.add_exception_handler(AppError,                app_error_handler)
app.add_exception_handler(Exception,               unhandled_exception_handler)

app.include_router(foods.router)
```

Handler registration order matters. FastAPI matches the most specific type first — subclasses of `AppError` are caught by `app_error_handler` before falling through to the generic `Exception` handler.

```
food_tracker/
■■■ alembic/
■■■ alembic.ini
■■■ app/
■   ■■■ __init__.py
■   ■■■ main.py          ← updated: four exception handlers registered
■   ■■■ config.py
■   ■■■ database.py
■   ■■■ models.py
■   ■■■ schemas.py
■   ■■■ crud.py
■   ■■■ dependencies.py  ← updated: get_valid_food raises NotFoundError
■   ■■■ errors.py        ← NEW: ErrorResponse, AppError subclasses, handlers
■   ■■■ notifications.py
■   ■■■ routers/
■       ■■■ __init__.py
■       ■■■ foods.py
■■■ food_tracker.db
■■■ requirements.txt
```

1. Trigger each error type from `/docs` and confirm the envelope shape:

- `GET /foods/99999` → 404 `not_found`
- `POST /foods` with missing `name` field → 422 `validation_error`
- `GET /foods?sort_by=bad_column` → 422 `validation_error`

2. Add a `DuplicateError` that extends `ConflictError`, and raise it in a hypothetical `create_food` path if a food item with the same name already exists. Add a uniqueness check to `crud.create_food`:

```
# In crud.py
def create_food(db: Session, food: FoodItemCreate) -> FoodItemModel:
    existing = db.query(FoodItemModel).filter(FoodItemModel.name == food.name).first()
    if existing:
        raise ConflictError(f"A food item named '{food.name}' already exists.")
    ...
```

3. Add a `request_id` field to `ErrorResponse` using Python's `uuid` module. Generate a UUID per request in the exception handlers and include it in the response. Log the same UUID with the error so you can correlate client-reported errors with server logs.

```
import uuid
request_id = str(uuid.uuid4())
# Add to ErrorResponse and log alongside the error
```

- **Uniform error envelope** — every error, whatever its source, returns the same `{status, error, details}` shape. Clients have one parser for all failures.
- **Custom `HTTPException` handler** reformats FastAPI's default `{"detail": ...}` into the envelope.
- **Custom validation handler** strips Pydantic internals — no URLs, no nested paths beyond the field name, no `input` echo.
- **Domain exception classes** (`NotFoundError`, `ConflictError`) carry their own status and error codes. Raise sites read like business logic, not HTTP plumbing.
- **The 500 handler** logs the full traceback internally and returns a safe generic message — stack traces never reach the client.
- **Handler registration order** — FastAPI matches the most specific type first; register `AppError` before `Exception`.

Lesson 15 covers authentication — validating JWT tokens, protecting routes with a real `get_current_user` dependency, and replacing the stub from Lesson 11 with a working implementation.

LESSON 15

FastAPI Lesson 15 — Authentication Basics

In Lesson 11 we added a `get_current_user` stub that always returned the same hardcoded user. This lesson replaces it with a real implementation: a `User` model stored in the database, password hashing with `passlib`, JWT tokens issued on login, and an `OAuth2PasswordBearer` scheme that extracts and validates the token on every protected request.

Passwords

Passwords are never stored in plain text. `passlib` runs the password through `bcrypt` — a one-way hash. Login verification hashes the candidate password and compares hashes; the original password is never recoverable from the stored value.

JWT (JSON Web Token)

A JWT is a signed string containing a JSON payload. The server signs it with a secret key; any server that knows the secret can verify the signature without querying the database. A typical payload:

```
{ "sub": "1", "exp": 1735000000 }
```

`sub` (subject) is the user ID. `exp` is the expiry Unix timestamp. The client sends the token in the `Authorization: Bearer <token>` header on every request.

OAuth2PasswordBearer

FastAPI's `OAuth2PasswordBearer` is a dependency that reads the `Authorization: Bearer` header and returns the raw token string. It also tells `/docs` to show a login form, so you can test protected endpoints directly in the browser.

```
pip install python-jose[cryptography] passlib[bcrypt]
```

- `python-jose` — JWT encoding, decoding, and signature verification.
- `passlib[bcrypt]` — password hashing context.

Add a `User` ORM model to `app/models.py`:

```
# app/models.py (addition)
from sqlalchemy import Boolean, Column, Integer, String
from app.database import Base

class User(Base):
    __tablename__ = "users"

    id           = Column(Integer, primary_key=True, index=True)
    username     = Column(String, unique=True, index=True, nullable=False)
    email        = Column(String, unique=True, index=True, nullable=False)
    hashed_password = Column(String, nullable=False)
    is_active    = Column(Boolean, default=True)
    is_admin     = Column(Boolean, default=False)
```

Add schemas to `app/schemas.py`:

```
# app/schemas.py (additions)
class UserCreate(BaseModel):
    username: str
    email:    str
    password: str   # plain-text, validated here, never stored

class UserResponse(BaseModel):
    id:       int
    username: str
    email:    str
    is_admin: bool
    model_config = ConfigDict(from_attributes=True)

class TokenResponse(BaseModel):
    access_token: str
    token_type:   str = "bearer"
```

The `User` table is new — generate a migration instead of relying on `create_all`:

```
alembic revision --autogenerate -m "add users table"
alembic upgrade head
```

```python
# app/auth.py
from datetime import datetime, timedelta, timezone
from jose import JWTError, jwt
from passlib.context import CryptContext
from app.config import settings   # assumes SECRET_KEY, ALGORITHM, ACCESS_TOKEN_EXPIRE_MINUTES

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


def hash_password(plain: str) -> str:
    return pwd_context.hash(plain)


def verify_password(plain: str, hashed: str) -> bool:
    return pwd_context.verify(plain, hashed)


def create_access_token(subject: str | int) -> str:
    """Create a signed JWT with an expiry of ACCESS_TOKEN_EXPIRE_MINUTES."""
    expire = datetime.now(timezone.utc) + timedelta(
        minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES
    )
    payload = {"sub": str(subject), "exp": expire}
    return jwt.encode(payload, settings.SECRET_KEY, algorithm=settings.ALGORITHM)


def decode_access_token(token: str) -> str:
    """
    Decode and verify a JWT. Returns the subject (user id as string).
    Raises JWTError if the token is invalid or expired.
    """
    payload = jwt.decode(token, settings.SECRET_KEY, algorithms=[settings.ALGORITHM])
    subject = payload.get("sub")
    if subject is None:
        raise JWTError("Token has no subject.")
    return subject
```

Add the three new fields to `app/config.py`:

```python
# app/config.py
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    DATABASE_URL:               str = "sqlite:///./food_tracker.db"
    SECRET_KEY:                 str = "change-me-in-production"
    ALGORITHM:                  str = "HS256"
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30

    class Config:
        env_file = ".env"

settings = Settings()
```

In production, set `SECRET_KEY` to a long random value via an environment variable or `.env` file — never commit it to version control.

Generate a strong key:

```
python -c "import secrets; print(secrets.token_hex(32))"
```

```
# app/crud.py (additions)
from app.models import User as UserModel
from app.schemas import UserCreate
from app import auth

def get_user_by_username(db: Session, username: str) -> UserModel | None:
    return db.query(UserModel).filter(UserModel.username == username).first()

def get_user_by_id(db: Session, user_id: int) -> UserModel | None:
    return db.query(UserModel).filter(UserModel.id == user_id).first()

def create_user(db: Session, user_in: UserCreate) -> UserModel:
    db_user = UserModel(
        username        = user_in.username,
        email           = user_in.email,
        hashed_password = auth.hash_password(user_in.password),
    )
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
```

Replace the stub in `app/dependencies.py`:

```
# app/dependencies.py — updated get_current_user
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError
from sqlalchemy.orm import Session
from app import auth, crud
from app.database import get_db
from app.errors import AppError
from app.models import User as UserModel

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/token")


class AuthenticationError(AppError):
    status_code = 401
    error_code  = "unauthorised"
    message     = "Could not validate credentials."


def get_current_user(
    token: str     = Depends(oauth2_scheme),
    db:    Session = Depends(get_db),
) -> UserModel:
```

```
    try:
        user_id = auth.decode_access_token(token)
    except JWTError:
        raise AuthenticationError()

    user = crud.get_user_by_id(db, int(user_id))
    if user is None or not user.is_active:
        raise AuthenticationError()

    return user


def get_current_admin(user: UserModel = Depends(get_current_user)) -> UserModel:
    """Requires get_current_user to succeed AND user to be an admin."""
    if not user.is_admin:
        raise ForbiddenError("Admin access required.")
    return user
```

Update `CurrentUser` references in the router — `user` is now a full `UserModel` instance, so you can access `user.id`, `user.username`, `user.is_admin` directly. The `CurrentUser` dataclass from the stub can be removed.

```
# app/routers/auth.py
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from app import auth, crud
from app.database import get_db
from app.errors import ConflictError
from app.schemas import TokenResponse, UserCreate, UserResponse

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


@router.post("/register", response_model=UserResponse, status_code=status.HTTP_201_CREATED)
def register(user_in: UserCreate, db: Session = Depends(get_db)):
    if crud.get_user_by_username(db, user_in.username):
        raise ConflictError(f"Username '{user_in.username}' is already taken.")
    return crud.create_user(db, user_in)


@router.post("/token", response_model=TokenResponse)
def login(
    form: OAuth2PasswordRequestForm = Depends(),
    db:   Session                   = Depends(get_db),
):
    user = crud.get_user_by_username(db, form.username)
    if user is None or not auth.verify_password(form.password, user.hashed_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password.",
            headers={"WWW-Authenticate": "Bearer"},
        )
    token = auth.create_access_token(subject=user.id)
    return TokenResponse(access_token=token)
```

`OAuth2PasswordRequestForm` is a FastAPI built-in that parses the standard OAuth2 form fields (`username`, `password`) from a `application/x-www-form-urlencoded` body — exactly what the `/docs` login dialog sends.

Update `foods.py` to use the real dependency:

```
# app/routers/foods.py — updated imports
from app.dependencies import get_current_user, get_current_admin
from app.models import User as UserModel

# POST — any authenticated user
@router.post("", response_model=FoodItemResponse, status_code=status.HTTP_201_CREATED)
def create_food(
    food:         FoodItemCreate,
    background_tasks: BackgroundTasks,
    db:           Session   = Depends(get_db),
    current_user: UserModel = Depends(get_current_user),
):
    return crud.create_food(db, food)

# DELETE — admin only
@router.delete(
    "/{food_id}",
    status_code=status.HTTP_204_NO_CONTENT,
)
def delete_food(
    background_tasks: BackgroundTasks,
    food:         FoodItemModel = Depends(get_valid_food),
    db:           Session       = Depends(get_db),
    current_user: UserModel     = Depends(get_current_admin),
):
    item_id, item_name = food.id, food.name
    crud.delete_food(db, food.id)
    background_tasks.add_task(log_deletion, item_id, item_name, current_user.username)
```

```
# app/main.py
from app.routers import auth as auth_router, foods

app.include_router(auth_router.router)
app.include_router(foods.router)
```

---

1. Go to `http://localhost:8000/docs`.

2. Click **POST /auth/register** → create a user.
3. Click the **Authorize** button (top right) → enter username and password → click **Authorize**.
4. All subsequent requests in `/docs` include the `Authorization: Bearer <token>` header automatically.
5. Try `GET /foods/1` — it now requires authentication and returns a 401 if the token is missing or invalid.

```
food_tracker/
├── alembic/
├── alembic.ini
├── .env                        ← SECRET_KEY (never commit)
├── app/
│   ├── __init__.py
│   ├── main.py                 ← updated: auth router registered
│   ├── config.py               ← updated: SECRET_KEY, ALGORITHM, expiry
│   ├── database.py
│   ├── models.py               ← updated: User model added
│   ├── schemas.py              ← updated: UserCreate, UserResponse, TokenResponse
│   ├── crud.py                 ← updated: user CRUD functions
│   ├── auth.py                 ← NEW: hash_password, verify_password, JWT encode/decode
│   ├── dependencies.py         ← updated: real get_current_user, get_current_admin
│   ├── errors.py               ← updated: AuthenticationError added
│   ├── notifications.py
│   └── routers/
│       ├── __init__.py
│       ├── auth.py             ← NEW: /register, /token
│       └── foods.py            ← updated: uses real UserModel dependency
├── food_tracker.db
└── requirements.txt
```

1. Add a `GET /auth/me` endpoint that returns the currently authenticated user's profile:

```
@router.get("/me", response_model=UserResponse)
def get_me(current_user: UserModel = Depends(get_current_user)):
    return current_user
```

2. Add token expiry to the `TokenResponse`. Return `expires_in` (seconds) alongside `access_token`:

```
class TokenResponse(BaseModel):
    access_token: str
    token_type:   str = "bearer"
    expires_in:   int  # ACCESS_TOKEN_EXPIRE_MINUTES * 60
```

3. Protect `GET /foods` with `get_current_user` (read-only access). Leave `POST`, `PUT`, `PATCH`, `DELETE` protected. Verify in `/docs` that unauthenticated `GET /foods` now returns a 401.

4. Try decoding a valid token manually in the Python REPL to inspect the payload:

```
from app.auth import decode_access_token
user_id = decode_access_token("paste-your-token-here")
print(user_id)
```

- **Passwords are one-way hashed with bcrypt** — the original password is never stored or recoverable. `verify_password` hashes the candidate and compares hashes.
- **JWTs are signed, not encrypted** — the payload is base64-encoded and visible to anyone. Never put sensitive data in the payload. The signature proves the token was issued by your server.
- `OAuth2PasswordBearer` extracts the `Authorization: Bearer` header and tells `/docs` to show a login form.
- `decode_access_token` **raises** `JWTError` **on invalid or expired tokens** — the dependency catches it and raises `AuthenticationError` (401).
- `get_current_admin` **composes with** `get_current_user` — authentication runs first, then the admin check. Two concerns, two dependencies.
- `SECRET_KEY` **must be kept secret** — store it in `.env`, load it via `pydantic-settings`, and never commit it.

Lesson 16 covers testing with pytest — writing unit tests for crud functions and integration tests for routes using a test database, and verifying authenticated endpoints with a test token.

LESSON 16

FastAPI Lesson 16 — Testing with Pytest

A FastAPI application is straightforward to test because the framework ships with a synchronous `TestClient` built on `httpx`. You can drive full HTTP request-response cycles against your app in-process, without a running server, and swap the production database for an isolated in-memory SQLite database through dependency overrides. This lesson covers: pytest project setup, a reusable test database fixture, unit tests for CRUD functions, integration tests for routes, and testing authenticated endpoints.

```
pip install pytest httpx pytest-anyio
```

- `pytest` — the test runner.
- `httpx` — `TestClient` depends on it for async transport.
- `pytest-anyio` — only needed if you write async tests; skip if you stay synchronous (recommended for simplicity).

Add a `tests/` directory at the project root, parallel to `app/`:

```
food_tracker/
■■■ app/
■   ■■■ ...
■■■ tests/
■   ■■■ __init__.py
■   ■■■ conftest.py          ← shared fixtures
■   ■■■ test_crud.py         ← unit tests: crud functions against test DB
■   ■■■ test_foods.py        ← integration tests: food routes
■   ■■■ test_auth.py         ← integration tests: register, login, protected routes
■■■ pytest.ini               ← or pyproject.toml [tool.pytest.ini_options]
```

`pytest.ini` (minimal):

```
[pytest]
testpaths = tests
```

The central idea: override `get_db` so every test gets a fresh, isolated SQLite database. Nothing touches the production `food_tracker.db`.

```
# tests/conftest.py
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from app.database import Base
from app.main import app
from app.dependencies import get_db

# In-memory SQLite — destroyed when the connection closes
TEST_DATABASE_URL = "sqlite:///./test.db"

engine_test = create_engine(
    TEST_DATABASE_URL, connect_args={"check_same_thread": False}
)
TestingSessionLocal = sessionmaker(
    autocommit=False, autoflush=False, bind=engine_test
)


@pytest.fixture(scope="session", autouse=True)
def create_test_tables():
    """Create all tables once for the whole test session."""
    Base.metadata.create_all(bind=engine_test)
    yield
    Base.metadata.drop_all(bind=engine_test)


@pytest.fixture()
def db():
    """Yield a fresh, rolled-back database session per test."""
    connection = engine_test.connect()
    transaction = connection.begin()
    session = TestingSessionLocal(bind=connection)

    yield session

    session.close()
    transaction.rollback()
    connection.close()


@pytest.fixture()
def client(db):
    """TestClient with get_db overridden to use the test session."""
    def override_get_db():
        try:
            yield db
        finally:
            pass  # rollback handled by the db fixture

    app.dependency_overrides[get_db] = override_get_db
    with TestClient(app) as c:
        yield c
    app.dependency_overrides.clear()
```

Key design decisions:

- `scope="session"` on `create_test_tables` — the schema is created once and dropped at the end. This is fast.
- The `db` fixture wraps each test in a transaction that is rolled back after the test. Every test starts with a clean slate without recreating the schema.
- `dependency_overrides` is a plain dict on the app — setting `app.dependency_overrides[get_db]` replaces the dependency for the duration of the test and is cleared afterwards.

Unit tests call CRUD functions directly with the test session — no HTTP, no routing, no middleware.

```
# tests/test_crud.py
import pytest
from app import crud
from app.schemas import FoodItemCreate, UserCreate
from datetime import date


def test_create_food(db):
    food_in = FoodItemCreate(
        name="Milk",
        category="dairy",
        expiry_date=date(2026, 12, 31),
    )
    food = crud.create_food(db, food_in)

    assert food.id is not None
    assert food.name == "Milk"
    assert food.category == "dairy"


def test_get_food_returns_none_for_missing(db):
    result = crud.get_food(db, food_id=99999)
    assert result is None


def test_delete_food(db):
    food_in = FoodItemCreate(
        name="Bread",
        category="bakery",
        expiry_date=date(2026, 6, 1),
    )
    food = crud.create_food(db, food_in)
    food_id = food.id

    crud.delete_food(db, food_id)

    assert crud.get_food(db, food_id) is None


def test_create_user_hashes_password(db):
    user_in = UserCreate(username="alice", email="alice@example.com", password="secret123")
    user = crud.create_user(db, user_in)

    assert user.hashed_password != "secret123"
```

```
    assert user.hashed_password.startswith("$2b$")  # bcrypt prefix


def test_get_user_by_username(db):
    user_in = UserCreate(username="bob", email="bob@example.com", password="pw")
    crud.create_user(db, user_in)

    found = crud.get_user_by_username(db, "bob")
    assert found is not None
    assert found.username == "bob"

    missing = crud.get_user_by_username(db, "nobody")
    assert missing is None
```

Integration tests use the `client` fixture to send real HTTP requests through the full FastAPI stack.

```
# tests/test_foods.py
import pytest
from datetime import date


def test_list_foods_empty(client):
    response = client.get("/foods")
    assert response.status_code == 200
    data = response.json()
    assert data["items"] == []
    assert data["total"] == 0


def test_get_food_not_found(client):
    response = client.get("/foods/99999")
    assert response.status_code == 404
    body = response.json()
    # Uniform error envelope from Lesson 14
    assert body["error"] == "not_found"
    assert "details" in body


def test_create_food_requires_auth(client):
    response = client.post("/foods", json={
        "name": "Eggs",
        "category": "dairy",
        "expiry_date": "2026-12-31",
    })
    # No token — must be 401
    assert response.status_code == 401


def test_create_food_invalid_body(client, auth_headers):
    # Missing required field: expiry_date
    response = client.post("/foods", json={"name": "Eggs"}, headers=auth_headers)
    assert response.status_code == 422
    body = response.json()
    assert body["error"] == "validation_error"
    fields = [d["field"] for d in body["details"]]
    assert "expiry_date" in fields
```

Protected routes need a valid JWT in the `Authorization: Bearer` header. The cleanest approach is a fixture that registers a user, logs in, and returns the headers.

```
# tests/conftest.py (additions)
from app.auth import create_access_token
from app.models import User as UserModel
from app import crud
from app.schemas import UserCreate


@pytest.fixture()
def test_user(db):
    """Create and return a regular user for tests."""
    user_in = UserCreate(
        username="testuser",
        email="test@example.com",
        password="testpassword",
    )
    return crud.create_user(db, user_in)


@pytest.fixture()
def test_admin(db):
    """Create and return an admin user for tests."""
    user_in = UserCreate(
        username="adminuser",
        email="admin@example.com",
        password="adminpassword",
    )
    user = crud.create_user(db, user_in)
    user.is_admin = True
    db.commit()
    db.refresh(user)
    return user


@pytest.fixture()
def auth_headers(test_user):
    """Return Authorization headers for a regular user."""
    token = create_access_token(subject=test_user.id)
    return {"Authorization": f"Bearer {token}"}


@pytest.fixture()
def admin_headers(test_admin):
    """Return Authorization headers for an admin user."""
    token = create_access_token(subject=test_admin.id)
    return {"Authorization": f"Bearer {token}"}
```

With these fixtures, tests become very readable:

```
# tests/test_foods.py (additions)
from datetime import date


def test_create_food_authenticated(client, auth_headers):
    response = client.post("/foods", json={
```

```
        "name": "Yogurt",
        "category": "dairy",
        "expiry_date": "2026-12-31",
    }, headers=auth_headers)
    assert response.status_code == 201
    body = response.json()
    assert body["name"] == "Yogurt"
    assert "id" in body


def test_delete_food_requires_admin(client, auth_headers, admin_headers, db):
    from app import crud
    from app.schemas import FoodItemCreate
    # Create a food item to delete
    food = crud.create_food(db, FoodItemCreate(
        name="Old Cheese",
        category="dairy",
        expiry_date=date(2024, 1, 1),
    ))

    # Regular user: 403
    response = client.delete(f"/foods/{food.id}", headers=auth_headers)
    assert response.status_code == 403

    # Admin: 204
    response = client.delete(f"/foods/{food.id}", headers=admin_headers)
    assert response.status_code == 204
```

```
# tests/test_auth.py
def test_register(client):
    response = client.post("/auth/register", json={
        "username": "newuser",
        "email": "new@example.com",
        "password": "strongpassword",
    })
    assert response.status_code == 201
    body = response.json()
    assert body["username"] == "newuser"
    assert "password" not in body          # plain password never returned
    assert "hashed_password" not in body   # hash never returned


def test_register_duplicate_username(client):
    payload = {"username": "dupeuser", "email": "a@example.com", "password": "pw"}
    client.post("/auth/register", json=payload)

    response = client.post("/auth/register", json={
        "username": "dupeuser",
        "email": "b@example.com",
        "password": "pw",
    })
    assert response.status_code == 409
    assert response.json()["error"] == "conflict"


def test_login(client):
    client.post("/auth/register", json={
        "username": "loginuser",
```

```
        "email": "login@example.com",
        "password": "mypassword",
    })
    response = client.post("/auth/token", data={
        "username": "loginuser",
        "password": "mypassword",
    })
    assert response.status_code == 200
    body = response.json()
    assert "access_token" in body
    assert body["token_type"] == "bearer"


def test_login_wrong_password(client):
    client.post("/auth/register", json={
        "username": "wpuser",
        "email": "wp@example.com",
        "password": "correct",
    })
    response = client.post("/auth/token", data={
        "username": "wpuser",
        "password": "wrong",
    })
    assert response.status_code == 401


def test_protected_route_with_invalid_token(client):
    response = client.post("/foods", json={
        "name": "Test",
        "category": "other",
    # ... (truncated for readability)
```

```
# Run all tests
pytest

# Verbose output
pytest -v

# Run a single file
pytest tests/test_auth.py

# Run a single test
pytest tests/test_foods.py::test_create_food_authenticated

# Stop on first failure
pytest -x

# Show local variables in failures
pytest -l
```

Sample output:

```
tests/test_auth.py .......                                    [  7%]
tests/test_crud.py .....                                      [ 33%]
tests/test_foods.py .......                                   [100%]

19 passed in 0.84s
```

- **Third-party library internals** — do not test that `jwt.encode` works; test that your `create_access_token` returns a decodable string.
- **SQLAlchemy ORM mechanics** — do not test that `db.commit()` persists data; test that your CRUD function returns the expected record.
- **Framework routing** — do not test that FastAPI dispatches `GET /foods` to the right function; test that the function returns the expected response.

Focus tests on your own logic: business rules, error conditions, auth guards, and response shapes.

```
food_tracker/
■■■ alembic/
■■■ alembic.ini
■■■ .env
■■■ app/
■   ■■■ ... (unchanged)
■■■ tests/                  ← NEW
■   ■■■ __init__.py
■   ■■■ conftest.py         ← fixtures: db, client, auth_headers, admin_headers
■   ■■■ test_crud.py        ← CRUD unit tests
■   ■■■ test_foods.py       ← food route integration tests
■   ■■■ test_auth.py        ← auth route integration tests
■■■ pytest.ini              ← NEW
■■■ food_tracker.db
■■■ requirements.txt
```

---

1. Add a test that confirms the `GET /foods` response envelope contains `has_more: false` when all results fit in the default page. Then add enough food items to exceed the default limit and assert `has_more: true`.

2. Write a test for the expiring-soon background task side-effect: after creating a food item with an expiry date in the past, read the audit log file (or a log table if you added one in Lesson 13) and assert the deletion was logged.

3. Add a `pytest.mark.parametrize` test that sends invalid `expiry_date` values and confirms each returns a 422 with `"field": "expiry_date"` in the details:

```
@pytest.mark.parametrize("bad_date", ["not-a-date", "2024-13-01", "", 12345])
def test_create_food_invalid_date(client, auth_headers, bad_date):
    response = client.post("/foods", json={
        "name": "Test",
        "category": "other",
        "expiry_date": bad_date,
```

```
    }, headers=auth_headers)
    assert response.status_code == 422
```

- `TestClient` drives full HTTP request-response cycles in-process — no server required.
- `dependency_overrides` is the clean way to swap `get_db` for a test session without touching production code.
- **Transaction rollback per test** keeps tests isolated without recreating the schema between runs.
- **Token fixtures** (`auth_headers`, `admin_headers`) keep auth boilerplate out of individual tests.
- **Unit tests** call CRUD functions directly; **integration tests** use `client` to hit routes — both use the same test database session.
- **Test only your code** — not the framework, not third-party libraries.

Lesson 17 covers deployment — packaging the application with `uvicorn`, configuring for production, and containerising with Docker.

LESSON 17

FastAPI Lesson 17 — Deployment

Running `uvicorn app.main:app --reload` is fine for development, but `--reload` burns CPU watching files for changes and is not safe in production. This lesson covers: the right uvicorn flags for production, using Gunicorn with uvicorn workers to handle multiple concurrent requests, production configuration through environment variables, adding a health check endpoint, configuring CORS, writing a Dockerfile, composing the app with Docker Compose, and a pre-deploy checklist.

The minimal production uvicorn command drops `--reload` and specifies workers:

```
uvicorn app.main:app --host 0.0.0.0 --port 8000 --workers 4
```

Flag reference:

Flag	Value	Reason
`--host 0.0.0.0`	bind to all interfaces	required inside Docker or on a remote server
`--port 8000`	listen port	default; change if a proxy sits in front
`--workers 4`	process count	rule of thumb: 2 × CPU cores + 1
`--no-access-log`	omit	optional; reduces log noise if a reverse proxy logs instead
`--log-level warning`	warning	less verbose in production; errors still surface

Do not use `--reload` in production. It forks a file-watcher process and can cause high CPU usage or stale module state.

For heavier production workloads, use Gunicorn as the process manager and give it uvicorn workers. Gunicorn handles signals (graceful restart, worker recycling) while uvicorn handles the ASGI protocol.

```
pip install gunicorn
```

```
gunicorn app.main:app \
  --worker-class uvicorn.workers.UvicornWorker \
  --workers 4 \
  --bind 0.0.0.0:8000 \
  --timeout 60 \
  --access-logfile -
```

`--worker-class uvicorn.workers.UvicornWorker` — this is the key flag. It tells Gunicorn to spawn uvicorn processes instead of its default synchronous workers.

`--timeout 60` — kills and replaces workers that don't respond within 60 seconds. Prevents stuck workers from blocking requests.

`--access-logfile -` — writes access logs to stdout (standard for Docker).

For simple single-server deployments `uvicorn --workers` is sufficient. Prefer Gunicorn when you need worker recycling, graceful restarts, or integration with a process supervisor like systemd.

Production configuration lives in environment variables — never in source code. `pydantic-settings` loads them automatically.

```
# app/config.py
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    DATABASE_URL:               str = "sqlite:///./food_tracker.db"
    SECRET_KEY:                 str = "change-me-in-production"
    ALGORITHM:                  str = "HS256"
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30
    ALLOWED_ORIGINS:            str = "*"        # comma-separated for CORS
    DEBUG:                      bool = False

    class Config:
        env_file = ".env"

settings = Settings()
```

In production, override these via real environment variables (not the `.env` file):

```
export DATABASE_URL="postgresql+psycopg2://user:pass@db:5432/food_tracker"
export SECRET_KEY="$(python -c 'import secrets; print(secrets.token_hex(32))')"
export ACCESS_TOKEN_EXPIRE_MINUTES=60
export ALLOWED_ORIGINS="https://myapp.com,https://www.myapp.com"
```

The `.env` file is for local development only. Never commit it.

A health check endpoint lets your load balancer, Kubernetes, or Docker health check confirm the app is alive and the database is reachable.

```
# app/routers/health.py
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from sqlalchemy import text
from app.database import get_db

router = APIRouter(tags=["health"])


@router.get("/health")
def health(db: Session = Depends(get_db)):
    """Returns 200 if the app and database are healthy."""
    db.execute(text("SELECT 1"))   # confirms DB connection is live
    return {"status": "ok"}
```

Register it in `main.py`:

```
from app.routers import health
app.include_router(health.router)
```

A shallow health check (no DB query) is faster but misses database outages. Prefer the DB-ping version unless the latency matters.

If a browser-based frontend calls your API from a different origin, you need CORS middleware. Add it in `main.py`:

```
# app/main.py
from fastapi.middleware.cors import CORSMiddleware

allowed_origins = [o.strip() for o in settings.ALLOWED_ORIGINS.split(",")]

app.add_middleware(
    CORSMiddleware,
    allow_origins=allowed_origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

`allow_origins=["*"]` permits all origins — safe during development but too broad for production. Set `ALLOWED_ORIGINS` to your frontend's exact origin(s) in production:

```
ALLOWED_ORIGINS=https://myapp.com,https://www.myapp.com
```

`allow_credentials=True` is required if the browser sends cookies or an `Authorization` header.

SQLite is unsuitable for production — it does not handle concurrent writes. Switch to PostgreSQL:

```
pip install psycopg2-binary


# .env (dev) or environment variable (prod)
DATABASE_URL=postgresql+psycopg2://user:password@localhost:5432/food_tracker
```

No other application code changes: SQLAlchemy abstracts the driver. Run `alembic upgrade head` against the new database.

```
# Dockerfile
FROM python:3.12-slim

# System deps
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential libpq-dev \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Install Python deps first (Docker layer cache)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy app source
COPY . .

# Run as non-root
RUN adduser --disabled-password --gecos "" appuser
USER appuser

EXPOSE 8000

CMD ["gunicorn", "app.main:app", \
     "--worker-class", "uvicorn.workers.UvicornWorker", \
     "--workers", "4", \
     "--bind", "0.0.0.0:8000", \
     "--timeout", "60", \
     "--access-logfile", "-"]
```

Key Dockerfile decisions:

- `python:3.12-slim` — smaller than the full image; `build-essential` and `libpq-dev` are only needed to compile `psycopg2`.
- `COPY requirements.txt` **before** `COPY .` — Docker caches the pip install layer as long as `requirements.txt` hasn't changed. Changing source files doesn't invalidate this cache.
- **Non-root user** — running as `appuser` limits the damage if the application is compromised.

Generate from the active environment:

```
pip freeze > requirements.txt
```

Minimum for this project:

```
fastapi
uvicorn[standard]
gunicorn
sqlalchemy
alembic
pydantic-settings
python-jose[cryptography]
passlib[bcrypt]
psycopg2-binary
httpx
```

For local development or single-server deployment, Docker Compose runs the app and database together:

```
# docker-compose.yml
version: "3.9"

services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER:     food
      POSTGRES_PASSWORD: food
      POSTGRES_DB:       food_tracker
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U food"]
      interval: 5s
      timeout: 5s
      retries: 5

  app:
    build: .
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgresql+psycopg2://food:food@db:5432/food_tracker
      SECRET_KEY:   change-me-in-production
      ALLOWED_ORIGINS: "http://localhost:3000"
    depends_on:
      db:
        condition: service_healthy
    command: >
      sh -c "alembic upgrade head &&
             gunicorn app.main:app
```

```
            --worker-class uvicorn.workers.UvicornWorker
            --workers 4
            --bind 0.0.0.0:8000
            --timeout 60
            --access-logfile -"

volumes:
  postgres_data:
```

Start everything:

```
docker compose up --build
```

The `depends_on: condition: service_healthy` ensures the app waits for Postgres to be ready before running migrations.

```
# Build the image
docker build -t food-tracker .

# Run with environment variables
docker run -p 8000:8000 \
  -e DATABASE_URL="postgresql+psycopg2://user:pass@host:5432/food_tracker" \
  -e SECRET_KEY="your-secret-key" \
  food-tracker
```

Before deploying to production:

Security

- [ ] `SECRET_KEY` is a random 32-byte hex value, set via environment variable — not hardcoded
- [ ] `DEBUG=False`
- [ ] `ALLOWED_ORIGINS` is set to your frontend's exact origin — not `*`
- [ ] `.env` is in `.gitignore` and not committed
- [ ] The app runs as a non-root user in Docker

Database

- [ ] `DATABASE_URL` points to PostgreSQL, not SQLite
- [ ] `alembic upgrade head` has been run against the production database
- [ ] Database credentials are not in source code

Application

- [ ] `--reload` flag is absent
- [ ] Worker count is appropriate for the server's CPU count

- [ ] `GET /health` returns 200 and pings the database
- [ ] `requirements.txt` is current (`pip freeze > requirements.txt`)

Testing

- [ ] `pytest` passes with no failures
- [ ] Tests run against the test database, not production

```
food_tracker/
■■■ alembic/
■■■ alembic.ini
■■■ .env                    ← dev only — never commit
■■■ .gitignore              ← includes .env, __pycache__, *.db
■■■ Dockerfile              ← NEW
■■■ docker-compose.yml      ← NEW
■■■ requirements.txt        ← updated: gunicorn, psycopg2-binary
■■■ pytest.ini
■■■ app/
■   ■■■ __init__.py
■   ■■■ main.py             ← updated: CORS middleware, health router
■   ■■■ config.py           ← updated: ALLOWED_ORIGINS, DEBUG
■   ■■■ database.py
■   ■■■ models.py
■   ■■■ schemas.py
■   ■■■ crud.py
■   ■■■ auth.py
■   ■■■ dependencies.py
■   ■■■ errors.py
■   ■■■ notifications.py
■   ■■■ routers/
■       ■■■ __init__.py
■       ■■■ auth.py
■       ■■■ foods.py
■       ■■■ health.py       ← NEW
■■■ tests/
    ■■■ conftest.py
    ■■■ test_crud.py
    ■■■ test_foods.py
    ■■■ test_auth.py
```

---

1. Add a `GET /health/detailed` endpoint that returns database latency in milliseconds:

```
import time

@router.get("/health/detailed")
def health_detailed(db: Session = Depends(get_db)):
    start = time.monotonic()
    db.execute(text("SELECT 1"))
    latency_ms = round((time.monotonic() - start) * 1000, 2)
    return {"status": "ok", "db_latency_ms": latency_ms}
```

2. Add a `.dockerignore` file to exclude files from the Docker build context — this speeds up builds and reduces image size:

```
.env
.git
__pycache__
*.pyc
*.db
tests/
.pytest_cache/
```

3. Configure Gunicorn via a `gunicorn.conf.py` file instead of command-line flags. This is easier to version-control and extend:

```
# gunicorn.conf.py
import multiprocessing

worker_class   = "uvicorn.workers.UvicornWorker"
workers        = multiprocessing.cpu_count() * 2 + 1
bind           = "0.0.0.0:8000"
timeout        = 60
accesslog      = "-"
errorlog       = "-"
loglevel       = "warning"
```

Then run: `gunicorn app.main:app -c gunicorn.conf.py`

- **Drop `--reload` in production** — it forks a file-watcher and is never safe outside development.
- **Gunicorn + UvicornWorker** gives you a production-grade process manager with graceful restarts and worker recycling.
- `pydantic-settings` **loads from environment variables** — production configuration is set at the OS or container level, never in source code.
- **CORS is required for browser frontends** — restrict `allow_origins` to your actual frontend domain in production.
- **Switch to PostgreSQL** — SQLite does not handle concurrent writes and is unsuitable for production.
- **Run as non-root in Docker** — limits blast radius if the application is compromised.
- `GET /health` lets your infrastructure confirm the app and database are alive; make it the first URL your monitor pings.

You have built a production-ready FastAPI food expiration tracker covering:

1. Project setup and first route
2. Pydantic schemas and request validation

3. Path and query parameters
4. SQLAlchemy models and database sessions
5. Full CRUD operations
6. Alembic migrations
7. Response models and serialisation
8. Async routes
9. Middleware and lifespan events
10. Router organisation
11. Dependency injection with `Depends()`
12. Advanced filtering, pagination, and response envelopes
13. Background tasks
14. Uniform error handling
15. Authentication with JWT and bcrypt
16. Testing with pytest
17. Deployment with Docker and Gunicorn