Module 3: Accelerated Code Generation and Debugging, Lesson 2: API Endpoint Creation

Hands-On: API Endpoint Creation

Overview

This demo proves three claims from the lecture in real time:

- Contract-first prompts produce predictable, typed output — not creative guesses
- Documentation is a byproduct of well-structured prompts, not a separate task
- Keeping all prompts in one session eliminates context repetition

You will generate Pydantic schemas, FastAPI route handlers, and a full integration test suite using exactly three prompts, all in the same AI chat window. The student lab extends this to five endpoints with real JWT authentication.

Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Have a terminal ready with: pip install fastapi pydantic uvicorn httpx pytest pytest-asyncio
- Keep the five-part template visible on screen or a second monitor throughout
- Do NOT start a new chat between prompts — session continuity is the point

---

> **Remember**
>
> Every AI API prompt follows the same five parts. We will go on to prove it by building a Product Catalog API from scratch — schemas first, routes second, tests third — all in one session. Watch the pattern repeat.

During Hands-On Keep Visible if Possible

1. FRAMEWORK — what stack, version, constraints
2. DOMAIN — what the API models, field names, types
3. OPERATIONS — what actions (create / read / update / delete)
4. VALIDATION — business rules, ranges, required vs optional
5. BEHAVIOR — output format, what NOT to generate yet

Prompt 1 — Three Pydantic Schema Variants

What to Learn

Here's the five-part prompt in action. Tell the AI exactly what framework, what data, what operations, what rules — and most importantly tell it what NOT to generate yet. No routes. Just schemas.

Paste this into the AI chat:

```
FRAMEWORK: FastAPI with Pydantic v2. Python 3.11+.

DOMAIN: Product Catalog API. A product has these fields:
- id: UUID (auto-generated)
- name: str
- description: str
- price: Decimal
- stock_quantity: int
- category: enum — electronics | clothing | food | other
- is_active: bool (default True)
- created_at: datetime (auto-set)
- updated_at: datetime (auto-set on mutation)

OPERATIONS: Create, partial update, and retrieve products.

VALIDATION:
- price must be > 0
- stock_quantity >= 0
- name: 3-100 characters
- description: max 500 characters

BEHAVIOR: Generate ONLY three Pydantic model classes — no route
handlers, no database logic, no main.py. The three models are:
  1. ProductCreate  — fields required at creation time
  2. ProductUpdate  — all fields Optional for PATCH semantics
  3. ProductResponse — full model returned to callers, includes id,
     created_at, updated_at. Add model_config for JSON serialization
     of Decimal and datetime.
Add a docstring to each class explaining its purpose.
```

What you should get

Three Pydantic classes. ProductCreate has required fields. ProductUpdate wraps everything in Optional[...] = None. ProductResponse adds the id and timestamps. Field validators for price and name appear automatically.

You should ask:

"Why does ProductUpdate have every field as Optional?"

Answer: PATCH semantics — the client only sends what changed. Every None means 'leave this field alone.' This is contract-first thinking applied to update operations.

Prompt 2 — Route Handlers + Docstrings

> **Remember**
>
> Same chat window. The schemas are already in context — I don't repeat them. I just reference them and ask for routes. Watch the docstrings appear automatically because I specified them in the BEHAVIOR block.

Continue in the same chat session — do NOT start a new one:

```
Using the three Pydantic models above as your schema contract,
generate three FastAPI route handlers. Use an in-memory dict
`product_db: dict[UUID, ProductResponse] = {}` for storage.

OPERATIONS:
  POST   /products              — create product, return 201
  GET    /products/{product_id} — retrieve by UUID, 404 if missing
  PATCH  /products/{product_id} — partial update, requires auth

VALIDATION / ERROR BEHAVIOR:
  - Raise HTTPException(404) if product not found
  - Raise HTTPException(409) if a product with identical name exists
  - On PATCH, only overwrite fields that are not None in the request

AUTH: Add `current_user: str = Depends(get_current_user)` to the
PATCH route. Stub `get_current_user` as a function that reads an
`Authorization: Bearer <token>` header and raises HTTPException(401)
if missing. Do not implement JWT verification yet.

BEHAVIOR: Add a Google-style docstring to every route function.
Return type annotations required on all three functions.
```

> **What you should get**

> Three route functions with full type annotations. The docstrings describe args, returns, and raises — that's your OpenAPI documentation written for free. The get_current_user stub raises 401 on missing Authorization header.

Open http://localhost:8000/docs. Point to the three routes with request/response schemas pre-populated. We didn't write a single line of documentation. The docstrings in the route functions fed the OpenAPI generator automatically.

Tip: The 409 conflict error on the POST route appears in the Swagger UI because the docstring described it. This is the core payoff of contract-first — the docs reflect reality because they came from the same source as the code.

Prompt 3 — Integration Test Suite

> **What to learn**
>
> Still the same session. The AI knows the schemas. It knows the routes. Now I ask for tests — and I don't have to re-explain anything. This is the entire payoff of keeping your work in one context window.

Third prompt — still the same session:

```
Still in the same context. Generate a pytest integration test suite
for the three routes above using httpx.AsyncClient and pytest-asyncio.

FIXTURES:
  - `test_app` fixture: create a fresh FastAPI test app with an empty
    product_db per test (reset between tests)
  - `async_client` fixture: wrap the app with httpx.AsyncClient

TEST CASES (one function per case, descriptive names):
  1. POST /products with valid data → 201, all response fields present
  2. POST /products with price = -5 → 422 validation error
  3. POST /products twice with same name → 409 conflict
  4. GET /products/{id} for created product → 200, fields match
  5. GET /products/{nonexistent_id} → 404
  6. PATCH /products/{id} without Authorization header → 401
  7. PATCH /products/{id} with valid token → 200, only patched fields changed

No mocking of business logic. Use real HTTP calls through the test
client. Add a one-line comment above each test stating what behavior
it proves.
```

> **What you should get**
>
> A complete test file. Seven test functions with one-line comments. The fixtures handle setup and teardown. The AI generates 401/404/409 edge case tests automatically because the route docstrings described those error conditions.

Run the tests:

```
pytest test_products.py -v
```

At least one test may need a small fix. This is intentional:

- The AI wrote 90% in 30 seconds. You verify and own the 10%.
- If a test fails, read the error. The fix is usually one line.
- A failing test the AI wrote is still better than no test at all.

Wrap-Up — The Pattern, The Assignment Handoff

> **Remember**
>
> Three prompts. One session. We got typed schemas, documented routes, a working auth stub, and seven integration tests. The docs wrote themselves. The tests knew about the errors because the routes described them. That is contract-first thinking.

Key takeaways:

- Schemas before handlers, always
- Five-part prompt: framework, domain, operations, validation, behavior
- Session continuity eliminates repetition — the AI carries your context forward
- Documentation is a byproduct of clear BEHAVIOR blocks
- Tests live where the code lives — same session, same context