Lab M1.06 - API Calling with JSON Validation Report

Dina Bosma-Buczynska | 07.02.2026

How Pydantic validation works:

Pydantic works like a quality control inspector for data. First, you create a blueprint (called a model) that describes what good data should look like. For example, you must specify that a product name must be text, a price must be a number greater than zero, and a description must be at least 10 characters long. When someone sends you data, Pydantic automatically checks if it matches the blueprint. You can also add custom rules, like making sure a category is from your approved list (like "Electronics" or "Toys"). If something doesn't match, Pydantic stops everything and tells you exactly what's wrong, like "price must be positive" or "description is too short." This happens automatically before your code even tries to use the data.

Benefits of validation:

- Saves money by rejecting invalid data before calling expensive APIs.
- Prevents system crashes from unexpected data types or missing fields.
- Provides clear error messages that help users fix problems quickly.
- Automatically cleans data (trims spaces, removes duplicates, rounds numbers).
- Guarantees data quality throughout the entire workflow from input to output.

Challenges I faced:

ChatGPT response formatting: The API sometimes wrapped JSON in markdown code blocks (```json), which broke parsing until I added code to strip those markers

Figuring out validation rules: At first, I wasn't sure which validation constraints to use. For example, I had to learn that gt=0 means "greater than zero" (so 0.01 works but 0 doesn't), while ge=0 means "greater than or equal to zero" (so 0 is okay). I also struggled with deciding when to use required fields versus optional ones.

Improvements I made:

1. **Validating both input AND output:** Following the lab guide, I built two Pydantic models - one for validating the product data coming IN (ProductListing), and another for validating what ChatGPT sends back OUT (EnhancedProductListing). This double-check approach means I catch errors from both the client and the AI.

2. Testing with multiple JSON files: I created five test JSON files: one valid and four with different errors (negative price, wrong category, missing description, short name). This helped me verify that my validation actually catches problems before they break anything.

3. Building a complete workflow function: I combined everything into one function that does the full process: load JSON file, validate it, send to ChatGPT if valid, validate the response, and return the result. The lab emphasized checking input BEFORE calling the API to avoid wasting money on bad data.

4. Adding custom validators for data cleanup: I learned to use the @validator decorator to write custom rules. For example, my name validator strips extra spaces, my features validator removes duplicates and empty items, and my price validator rounds to two decimal places automatically. These improvements make the system handle messy real-world data better.