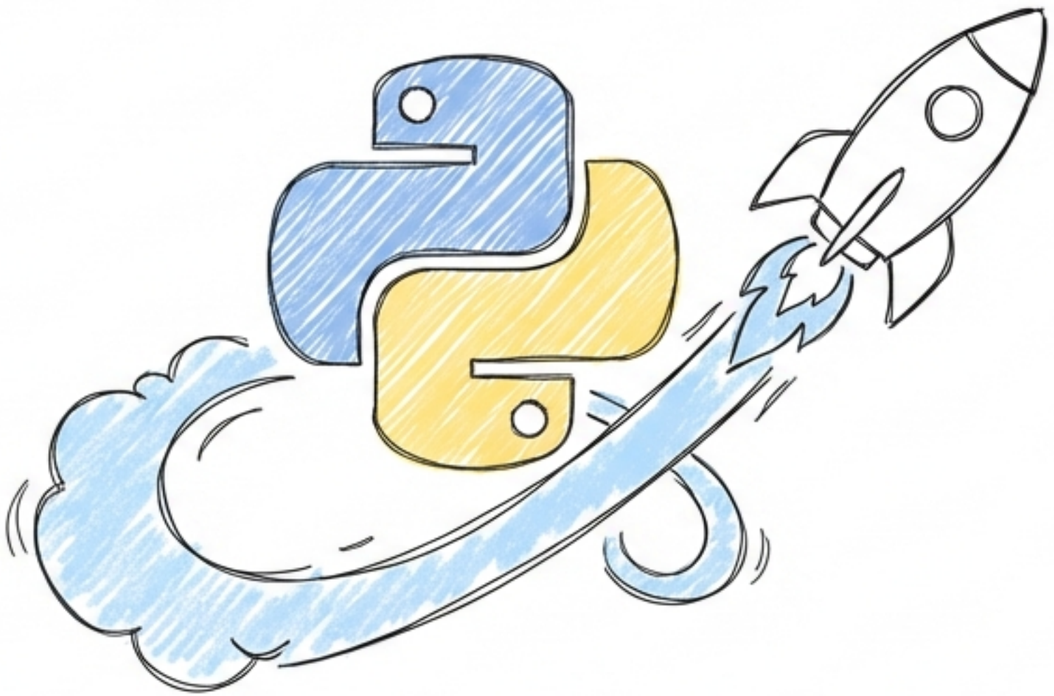
FastAPI for Beginners

Building Your First API, Step-by-Step



FastAPI for Beginners
A Visual Guide to Building Fast, Modern APIs
Senior Python Educator & Technical Visualizer

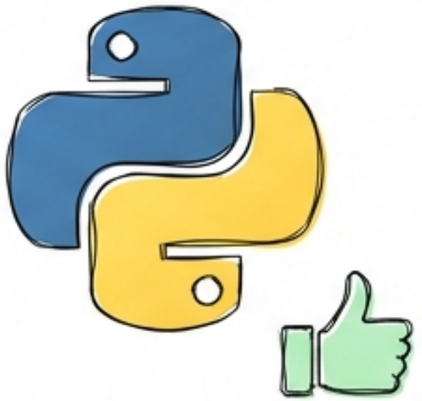
Why FastAPI?

High Performance:

One of the fastest Python frameworks, built for modern `async` web development.

Automatic Interactive Docs:

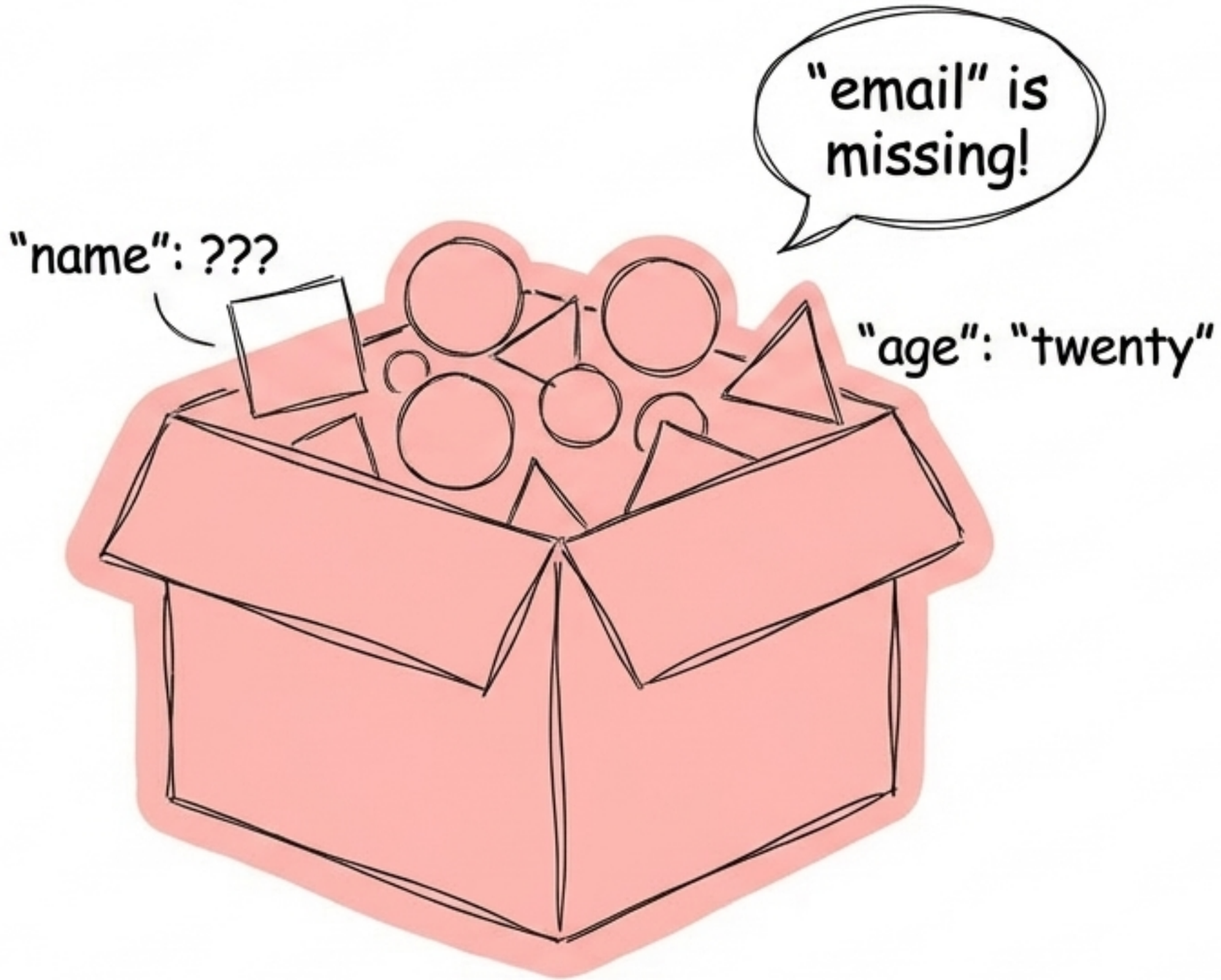
Generates a 'Swagger UI' for you to test your API live, just by visiting the `/docs` endpoint.



Modern Python:

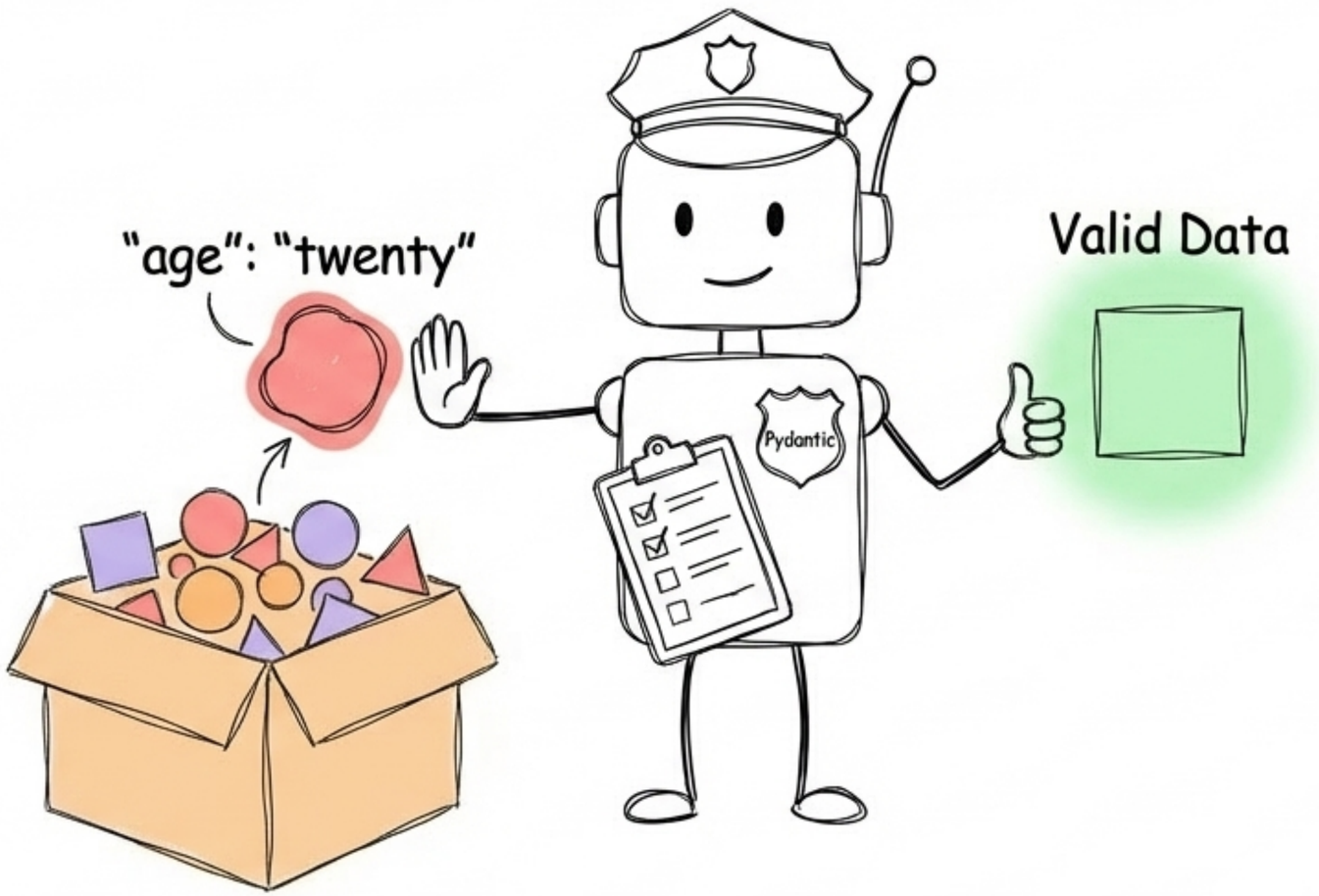
Uses standard Python type hints for clean, intuitive, and editor-friendly code.

The Problem with Raw Data



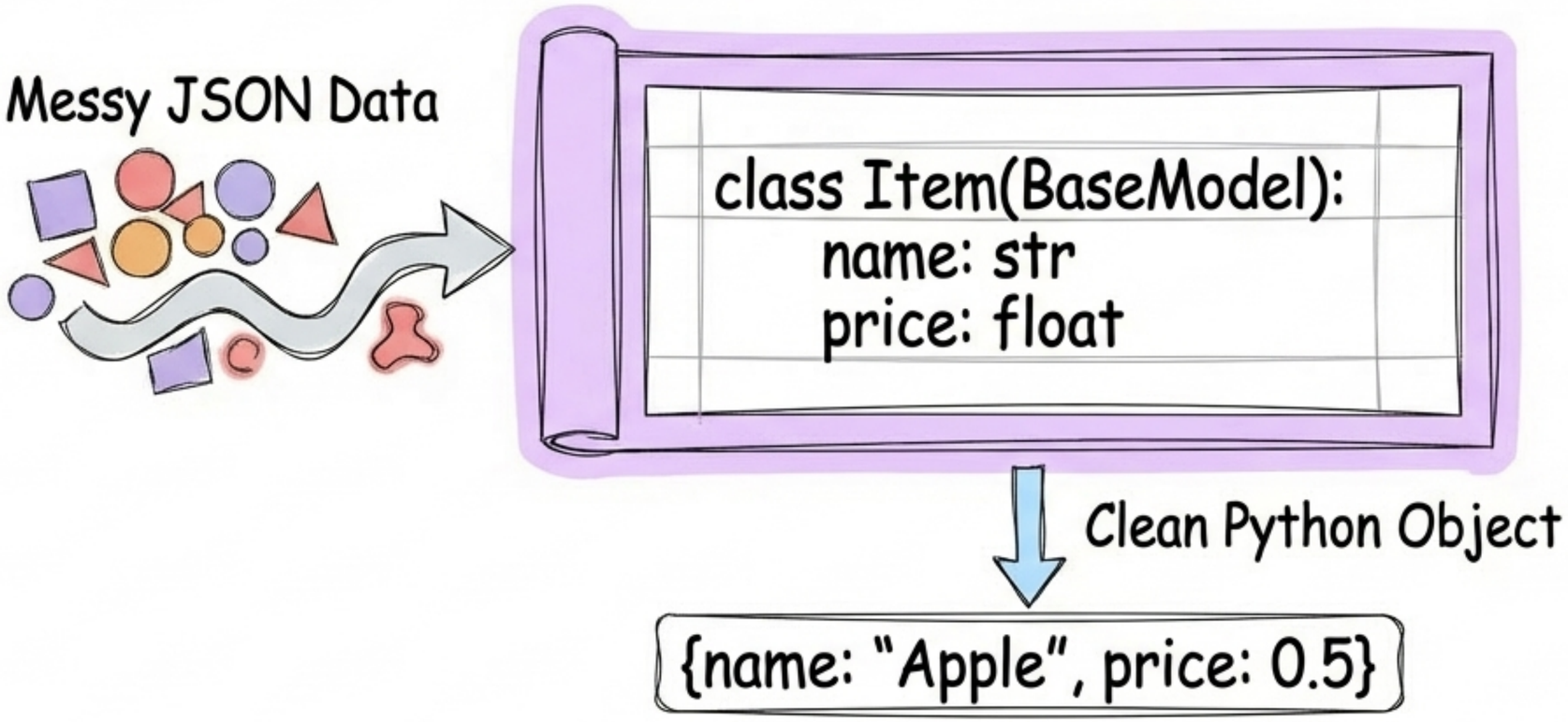
- Data from users can be unpredictable.
- A required field might be missing.
- A data type might be wrong (`age` sent as a string instead of a number).
- Handling this manually leads to buggy, complex code.

Enter Pydantic: The "Data Police"



- FastAPI uses a powerful library called **Pydantic** for data validation.
- It acts as a strict but helpful gatekeeper for your data.
- It enforces the "rules" you define for your API.
- If data is invalid, Pydantic rejects it automatically with a clear error message.

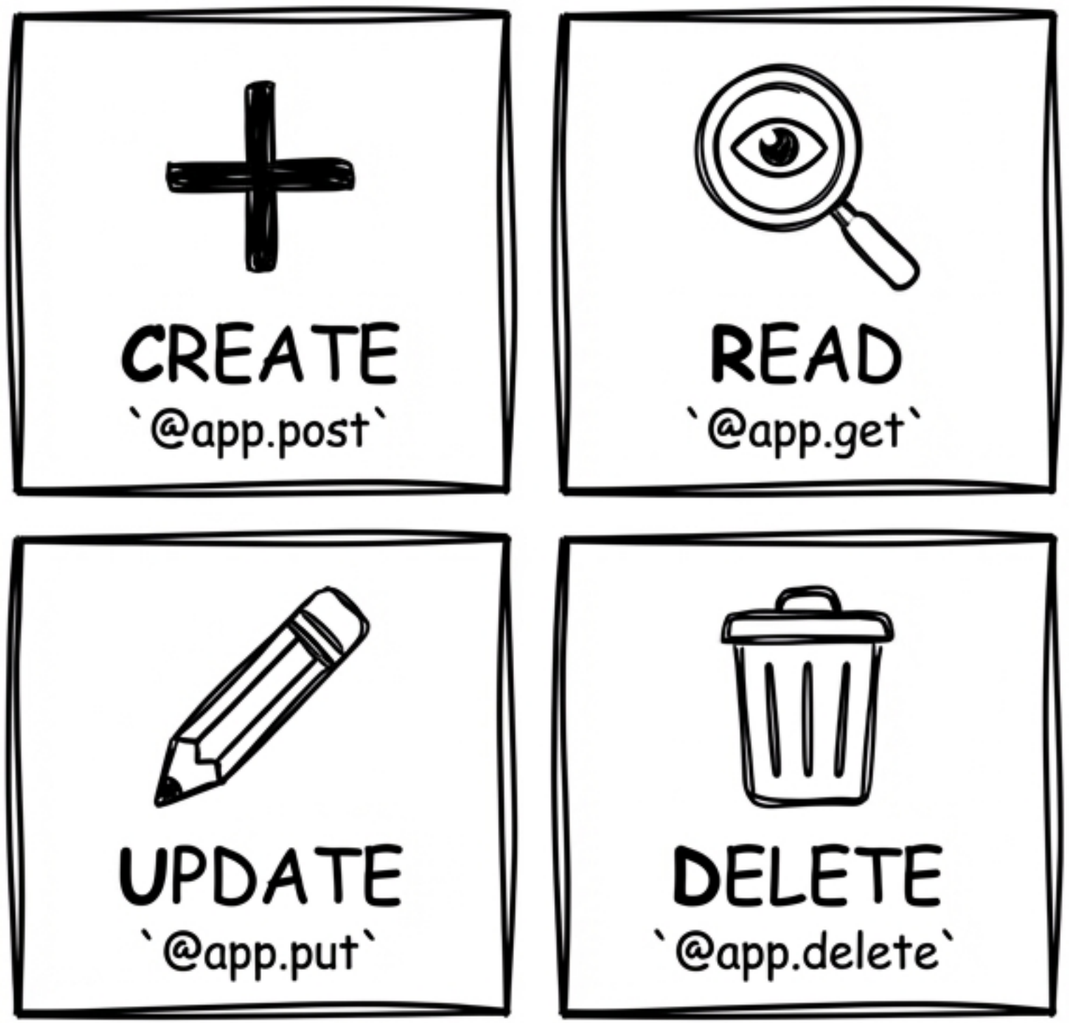
The `BaseModel`: Your Data Blueprint



- You define your data 'rules' by creating a class that inherits from 'BaseModel'.
- Use standard Python **type hints** (str, 'int, 'float, 'Optional', etc.).
- This class is your data's 'schema' or blueprint.

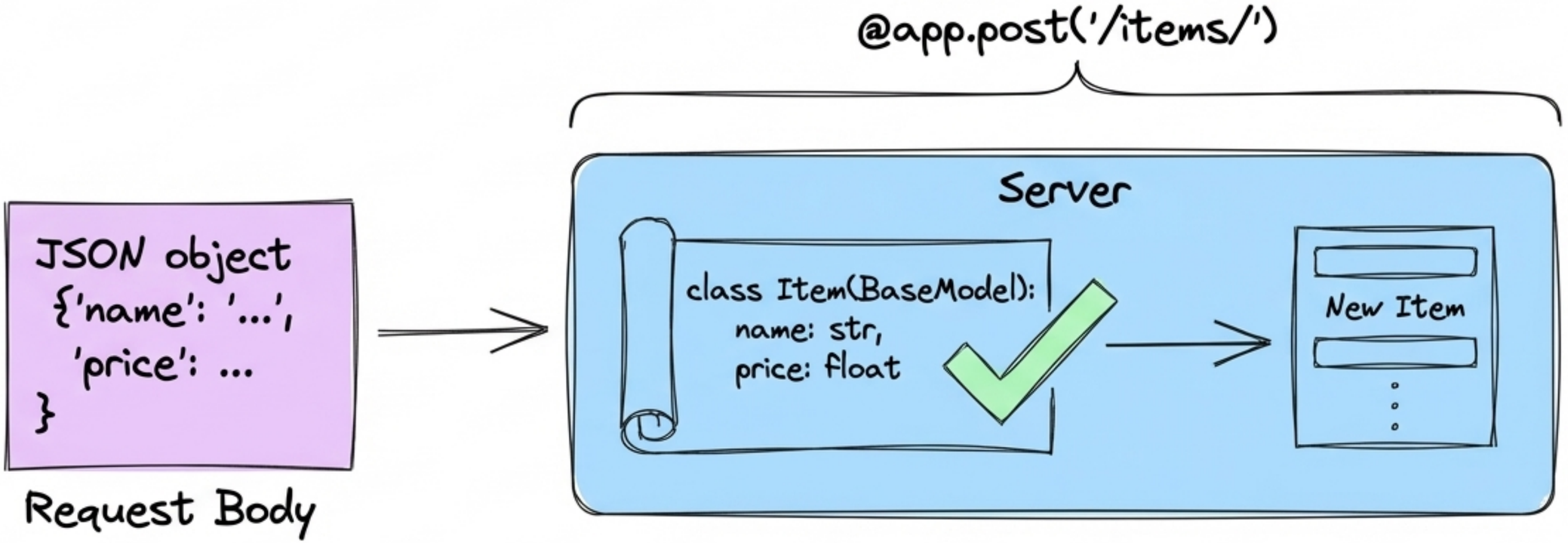
```
from pydantic import BaseModel
```

The Acronym: C.R.U.D.



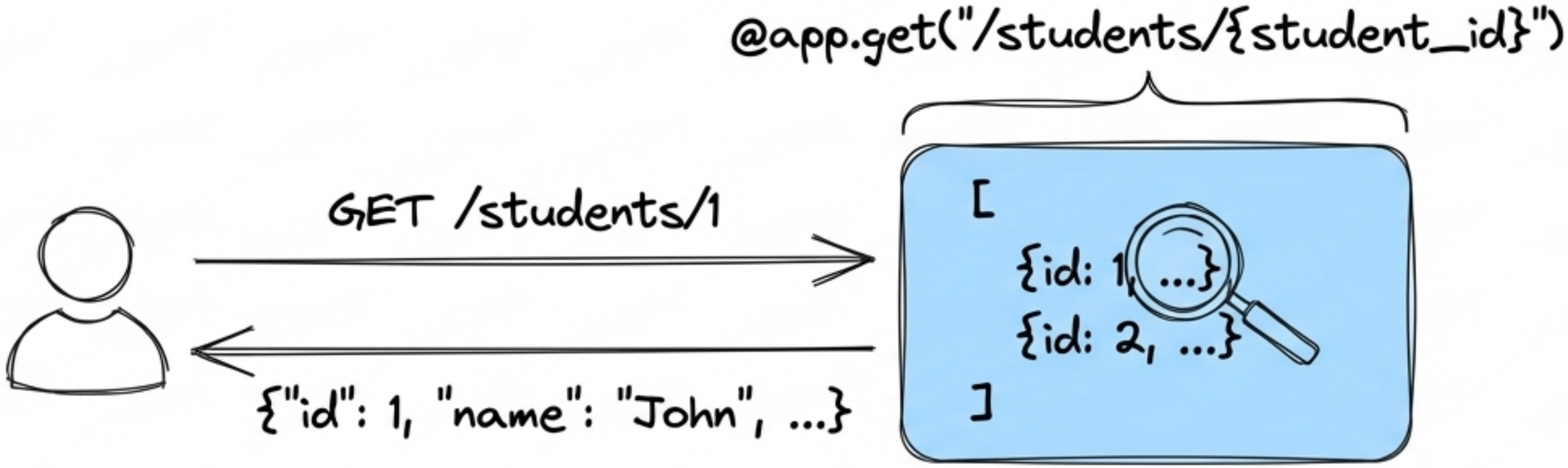
- Most APIs revolve around these four fundamental operations.
- **C**reate: Add new data.
- **R**ead: Retrieve existing data.
- **U**pdate: Modify existing data.
- **D**elete: Remove existing data.

CREATE (POST) with BaseModel



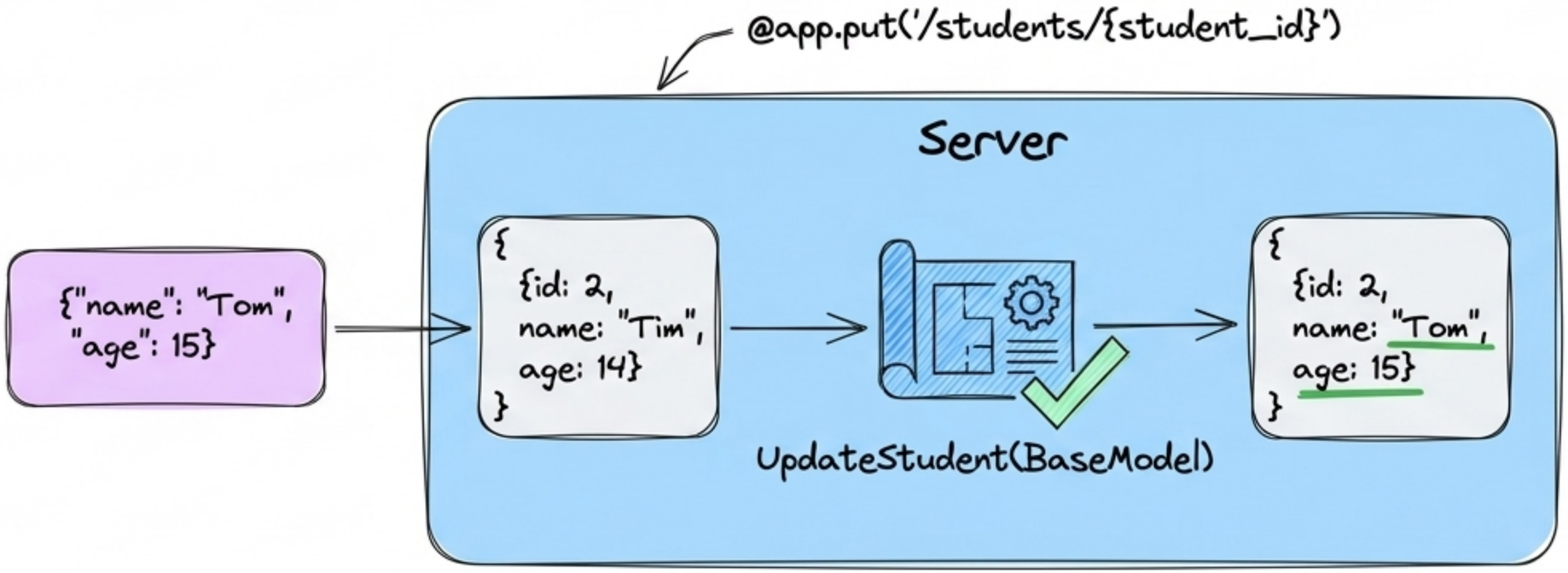
- Use the POST method to create new resources.
- Your endpoint function receives an argument typed with your BaseModel.
- FastAPI automatically validates the incoming JSON 'request body'.
- If valid, you receive a clean Python object to work with.

READ (GET)



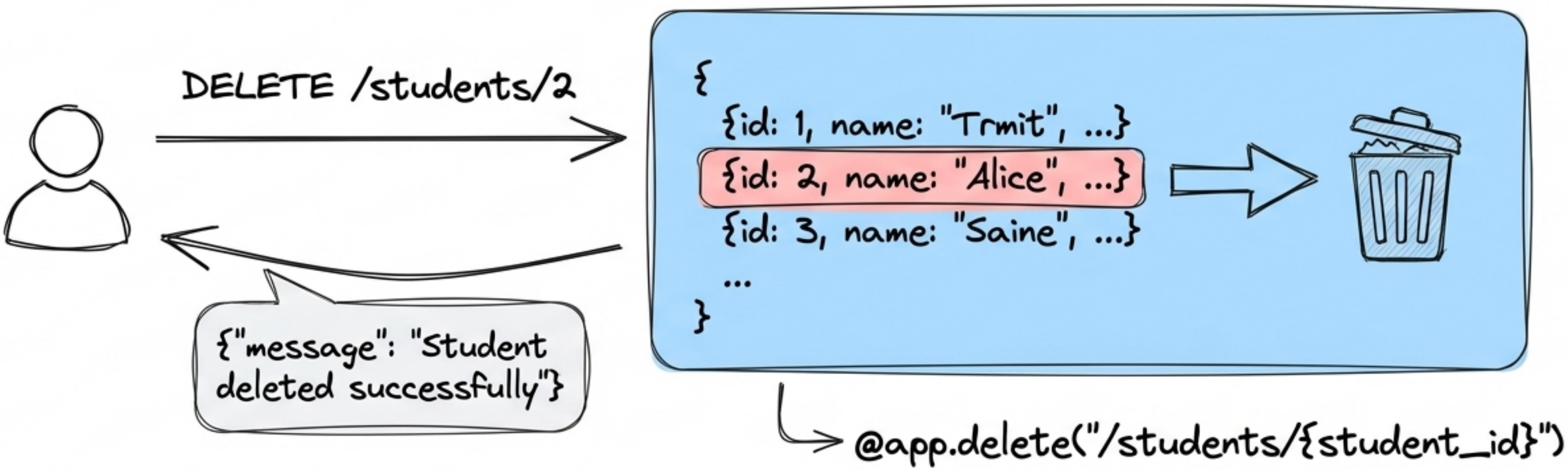
- Use the `GET` method to retrieve data.
- Fetch a specific item using a **path parameter** (e.g., `{student_id}`).
- The variable from the path is passed directly to your function.
- This operation should be ‘safe’—it should not change any data.

UPDATE (PUT) with BaseModel



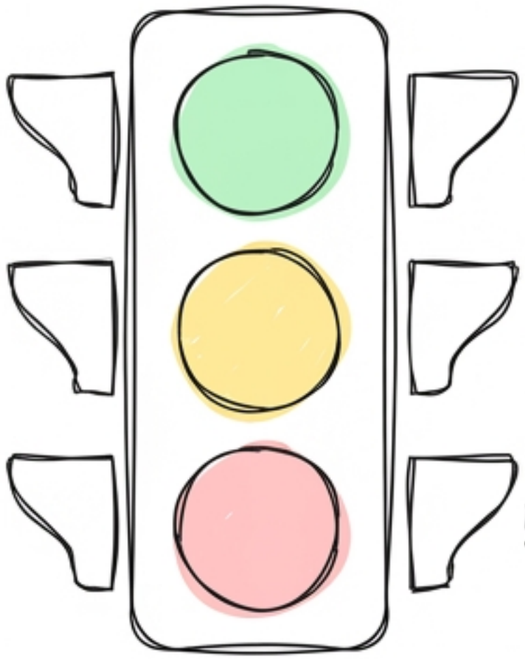
- Use the `PUT` method to update an existing resource.
- Identify the item using a path parameter (like the ID).
- Best practice: Use a separate `BaseModel` for updates with all fields marked as `Optional`.
- This prevents accidentally overwriting existing data with `null` if a user only sends one field to update.

DELETE



- Use the `DELETE` method to remove a resource.
- Identify the item to delete using a path parameter.
- Check if the item exists first to avoid errors.
- A successful deletion often returns a simple confirmation message.

Status Codes Overview



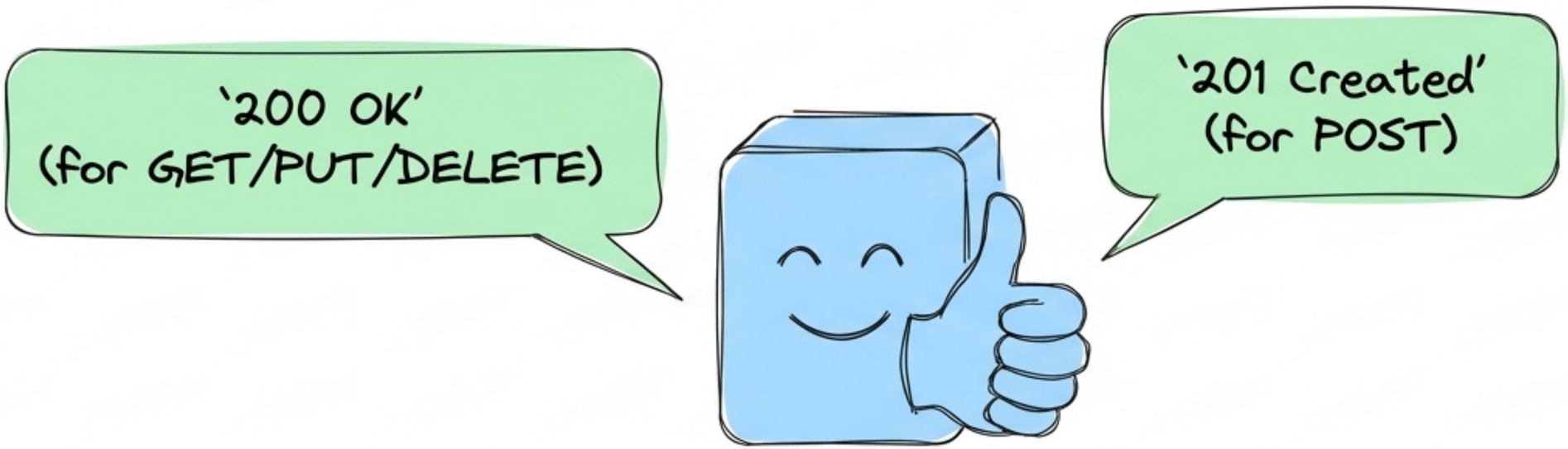
2xx: Success!

4xx: Your Fault (Client Error)

5xx: My Fault (Server Error)

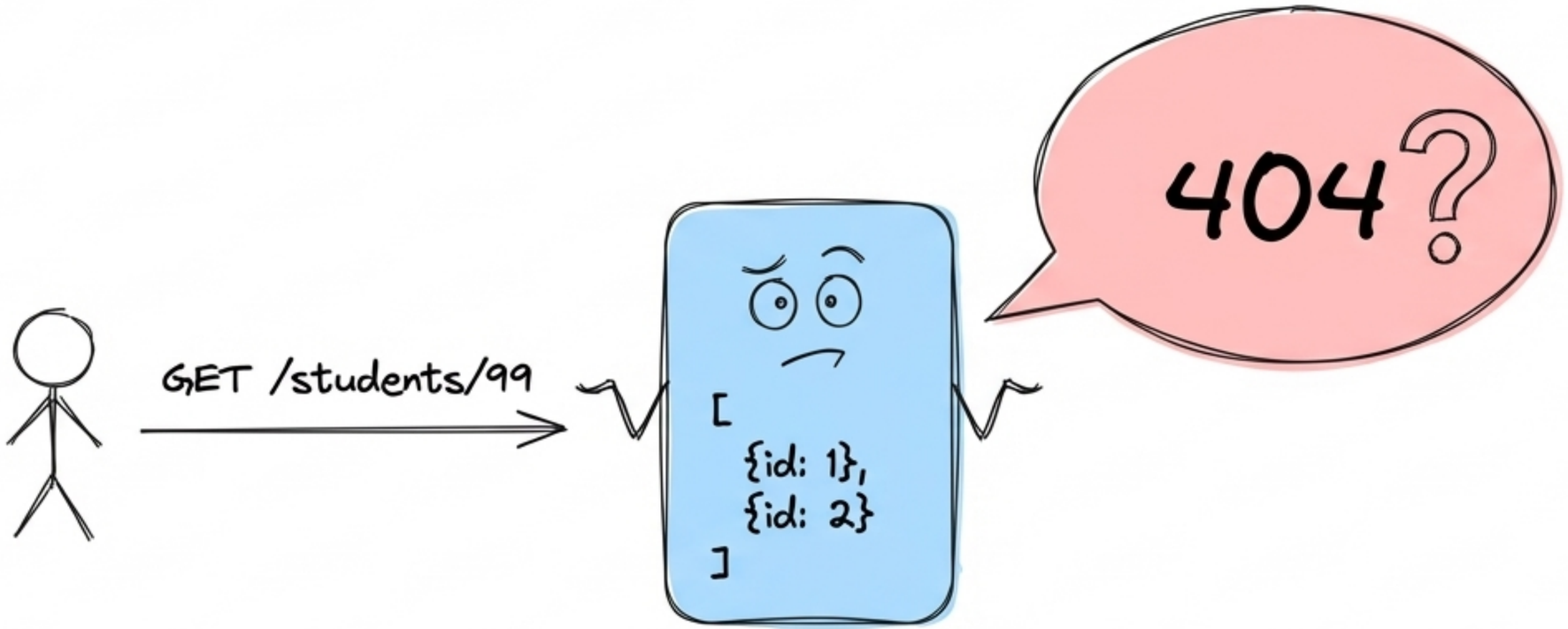
- HTTP status codes are how APIs communicate results.
- 2xx Range (Green): Everything went as planned. Success!
- 4xx Range (Yellow): The client sent a bad request (e.g., bad data, missing item).
- 5xx Range (Red): The server had an unexpected problem or bug.

The 200 Club: Success



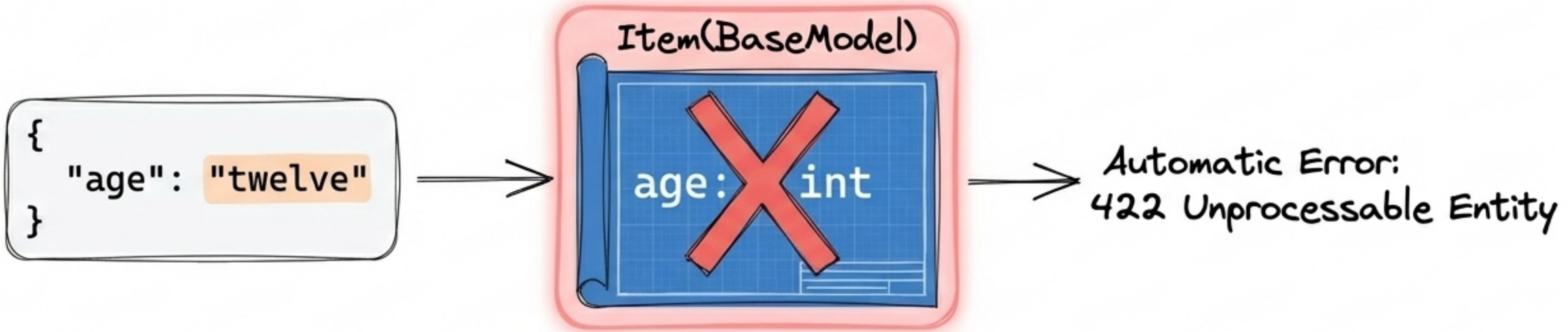
- '200 OK': The standard response for a successful 'GET', 'PUT', or 'DELETE'.
- '201 Created": Used specifically after a 'POST' request successfully creates a new resource.
- FastAPI is smart and uses these sensible defaults for you automatically.

404 Not Found: Handling Missing IDs



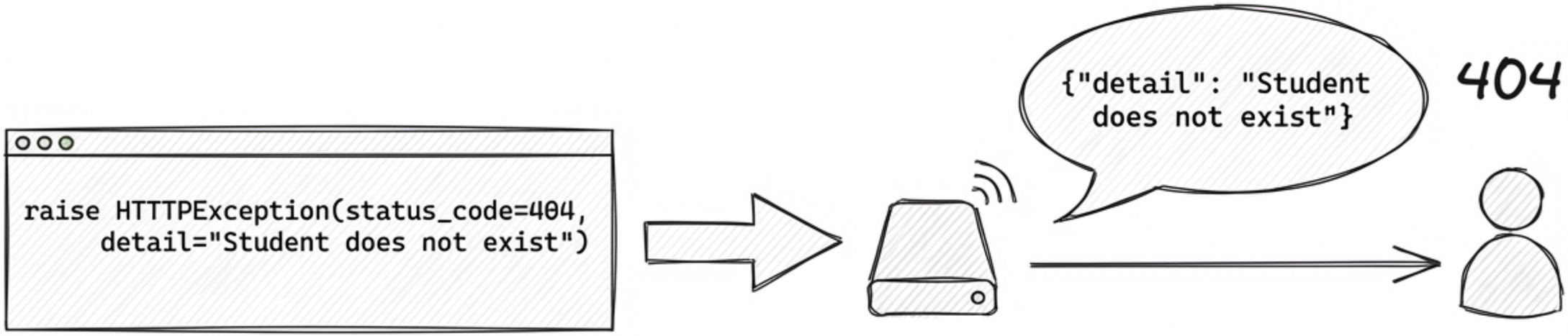
- This happens when a user requests an ID that doesn't exist.
- This is a client error, not a server bug.
- The correct response is `404 Not Found`.
- Your code must check if an item exists before trying to access or delete it.

422 Unprocessable Entity: Pydantic's Automatic Error



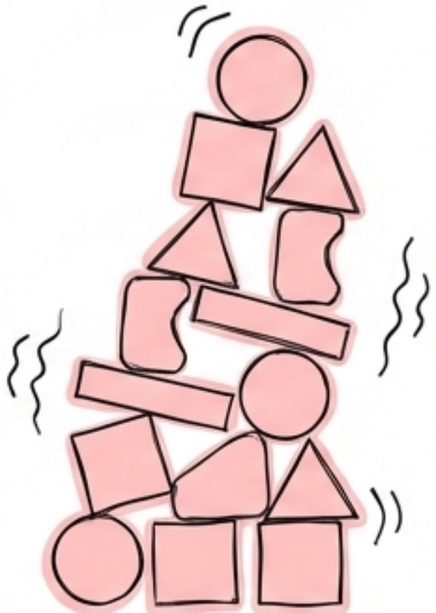
- This is Pydantic's automatic validation error.
- It's triggered when the JSON structure is right, but the data types are wrong.
- Example: Sending `"age": "twelve"` instead of `"age": 12`.
- FastAPI handles this for you! You write zero code for this check.

Raising Exceptions: The `HTTPException` Tool

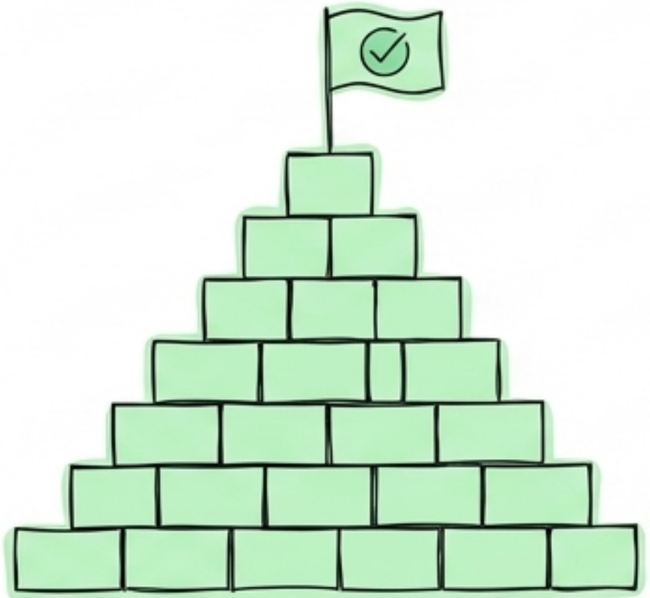


- For custom errors (like a 404), FastAPI gives you a tool: `HTTPException`.
- You can `raise` it anywhere in your code to stop execution.
- Specify the `status_code` and a helpful `detail` message.
- FastAPI catches the exception and sends the correct HTTP error response.

Why Test?



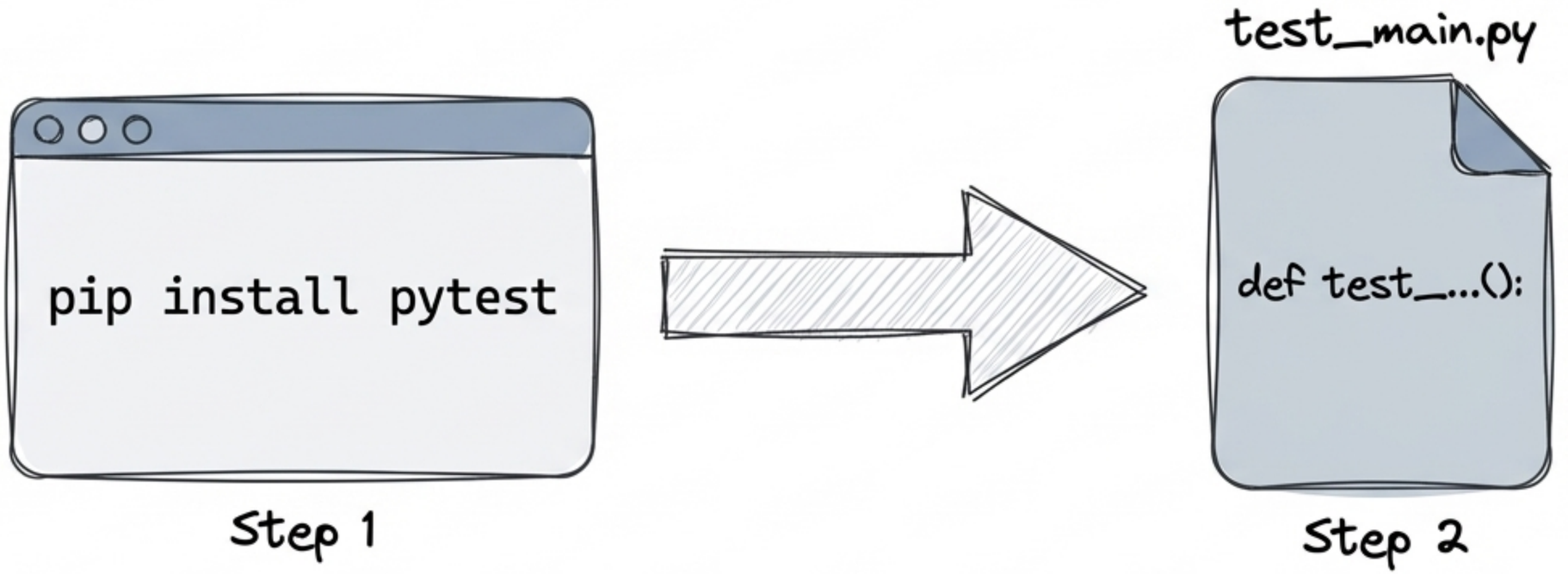
Untested Code



Tested Code

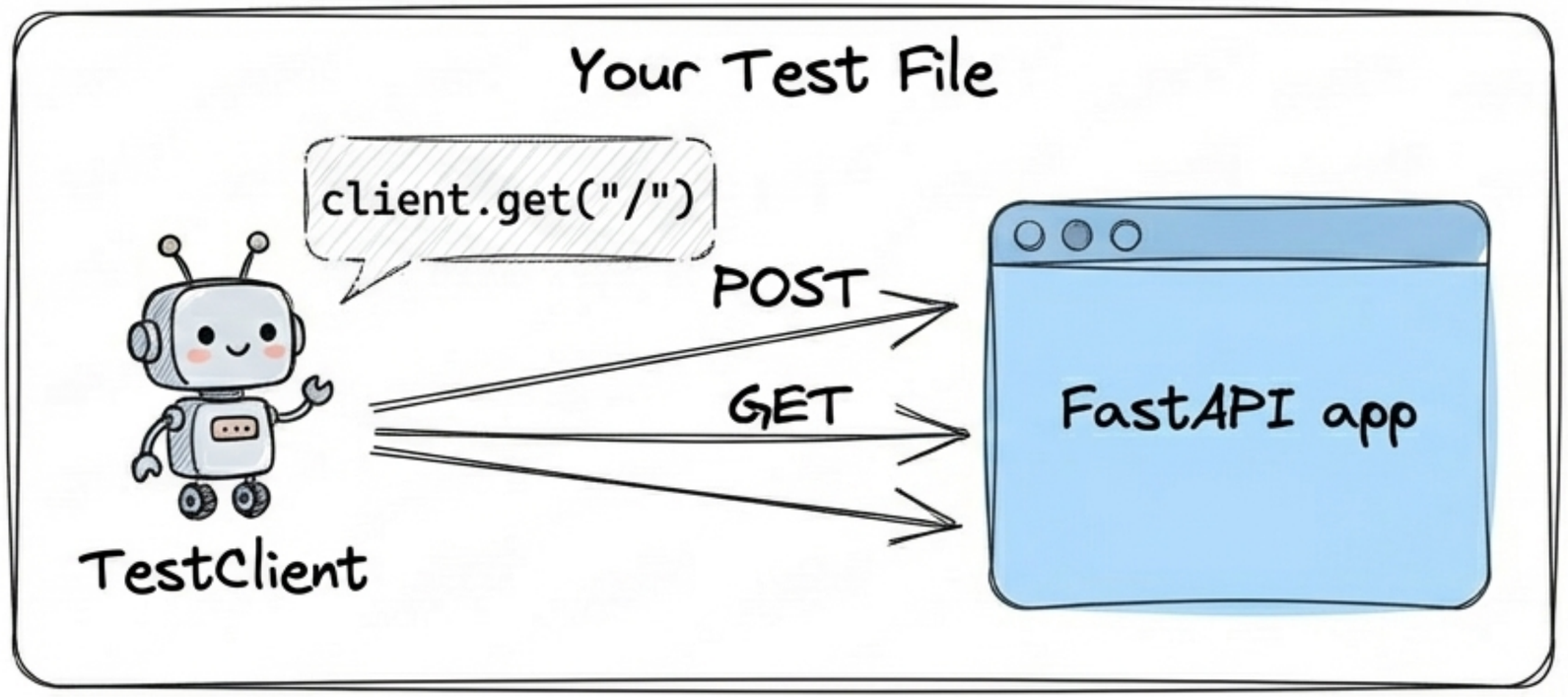
- Confidence: Prove your code works as you expect it to.
- Prevent Regressions: Ensure new changes don't break old features.
- Documentation: Tests act as living examples of how your API should behave.
- Testing gives you a safety net to refactor and improve your code.

Pytest Setup



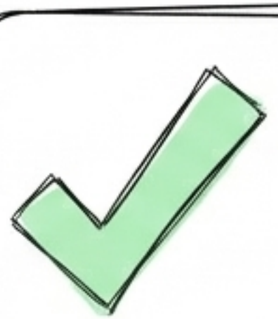
- We use a popular testing framework called `pytest`.
- Install it: `pip install pytest`.
- Create a new file for your tests (e.g., `test_main.py`).
- `pytest` automatically discovers and runs functions that start with `test_`.

The TestClient: Simulating Requests



- FastAPI provides a `TestClient` for simulating API requests in your tests.
- Import it: `from fastapi.testclient import TestClient`.
- It lets you call your endpoints directly from your Python test code.
- No need to run a live `uvicorn` server to test your application.

Writing a Test Case: The 'Happy Path'



1. `response = client.post("/students/...", json=...)`

2. `assert response.status_code == 201`

3. `assert response.json()["name"] == "Jerry"`

- A test function should focus on one specific behavior: does creating a user work?
- **Arrange:** Set up the data you plan to send.
- Add to post.
- **Act:** Call the endpoint using the `TestClient`.
- **Assert:** Check that the response is what you expected (the status code, the data inside).

Testing Errors: The 'Sad Path'

1. `response = client.post("/students/...", json={"age": "bad_data"})`

2. `assert response.status_code == 422`

- Just as important as testing success is testing for expected failures.
- Send a request you know should fail (e.g., a string for an integer field).
- Assert that you receive the correct error code (`422`, `404`, etc.).
- This ensures your error handling and validation are working correctly.