Week 2

Getting Started

Join the github classroom assignment with the following link: https://classroom.github.com/a/Cg-MTRGE

Once you join your repository will be created https://github.com/blueprint-learn/week2-techteam-sp26-{github-username}

Once you join, GitHub Classroom will automatically create your personal repository. You will receive an email confirming access.

After receiving access:

1. Clone your repository locally
2. Install dependencies
3. Run the starter FastAPI app

This repository contains the scaffolding you will extend during this week.

```
git clone https://github.com/blueprint-learn/week1-techteam-sp26-{github-username}.git
```

Recap last week

In Week 1, we established the core foundation of APIs using FastAPI. By the end of the session, you had:

- Built a running FastAPI server
- Created basic endpoints
- Used path parameters
- Verified correctness through automated CI tests

You also saw how an API defines a contract between a client and a server, and how tests enforce that contract.

Goal for today

By the end of this week, you will understand how APIs receive structured data and how FastAPI automatically validates incoming data.

This is a major step forward: your API will no longer only return data — it will safely accept data from clients.

Big idea

Last week, the interaction pattern was:

Client asks for data → Server returns data

This week, we expand that model:

Client sends data → Server validates and accepts data

This shift introduces one of the most important responsibilities of a backend API: validating inputs at the system boundary.

1. How Data Enters an API

There are three primary ways data reaches an API endpoint. Understanding these channels is essential for designing correct APIs.

Type	Where used	Example
Path	Identify resource	/users/3
Query	Filter/search	?limit=10
Body	Send data	POST JSON

Each serves a different purpose:

- Path parameters identify **which** resource
- Query parameters refine **how** to retrieve
- Request bodies provide **new data**

Path parameters

Path parameters help define a resource. They usually appear after a / . IDs are usually passed as path parameters to identify a resource.

```
/items/5
```

Query parameters

Query parameters appear after a ? in the URL and are commonly used for filtering, pagination, or search controls.

Examples to test:

```
/items?limit=10
```

```
@app.get("/items")
def list_items(limit: int = 10):
    return {"limit": limit}
```

FastAPI automatically:

- Applies default values when missing
- Converts types based on type hints
- Validates incorrect input

For example, if limit is defined as an integer and a string is provided, FastAPI will reject the request before your function executes.

Key insight:

FastAPI parses query strings into typed Python values.

Request body (JSON)

Real-world APIs must accept structured data. Common examples include:

- User signup information
- Orders or transactions
- Messages or posts
- Payment details

Clients send this data as JSON in the request body.

```
{
  "name": "Notebook",
  "price": 5.99
}
```

```
@app.post("/items")
def create_item(item: dict):
    return item
```

A naive implementation might accept this as a generic dictionary. However, this approach has critical problems:

- No validation
- No defined structure
- No guarantees about data shape

This leads to unstable and unsafe APIs

We need a schema.

4. Pydantic Models

FastAPI uses Pydantic models to define structured request bodies.

A Pydantic model specifies:

- Required fields
- Types
- Optional fields
- Validation rules

When used in an endpoint, FastAPI:

- Parses incoming JSON
- Validates types and structure
- Converts into a Python object
- Rejects invalid requests automatically

This provides:

- Schema definition
- Type enforcement
- Automatic parsing
- Validation

The model becomes the contract for accepted data.

Why Validation Matters

Validation at the API boundary protects the system from invalid or malicious input.

Without validation, systems risk:

- Database corruption
- Runtime crashes
- Security vulnerabilities
- Inconsistent data

With validation, APIs ensure:

- Safe inputs
- Predictable structure
- Stable behavior

Core principle:

Validation is defensive programming at the API boundary.

Define

```
from pydantic import BaseModel


class Item(BaseModel):
    name: str
    price: float
```

Use

```
@app.post("/items")
def create_item(item: Item):
    return item
```

Explain

- Schema definition
- Type enforcement
- Automatic parsing
- Validation

Revision #6
Created 18 February 2026 07:49:12 by Blueprint Admin
Updated 25 February 2026 17:33:04 by Blueprint Admin