Week 1

API & Backend Foundations with FastAPI

■ Learning Outcomes

- ✓ Understand backend fundamentals and the client-server-database flow
- ✓ Build basic APIs with FastAPI framework
- ✓ Implement data validation using Pydantic models
- ✓ Perform CRUD operations using in-memory data storage
- ✓ Configure CORS for frontend integration

Table of Contents

01 What is Backend?

Understanding the Server-Side of Web Applications

The backend is the "behind-the-scenes" part of a website or app, comprising the server, application logic, and database that power the user-facing (frontend) parts, handling data storage, security, and processing user requests.

■ Key Components & Functions

• **Server:** The hardware that receives requests from the frontend and sends back responses, managing data flow.

• **Application Logic:** The core code (e.g., Python, JavaScript) that processes data, implements business rules, and handles operations like logins.

• **Database:** Stores and organizes all the application's data, ensuring it's persistent, consistent, and secure (e.g., PostgreSQL).

• **APIs** (Application Programming Interfaces): Define how the frontend and backend communicate, acting as messengers for requests and data.

■■ How It Works

1. A user interacts with the frontend (e.g., clicks a button).

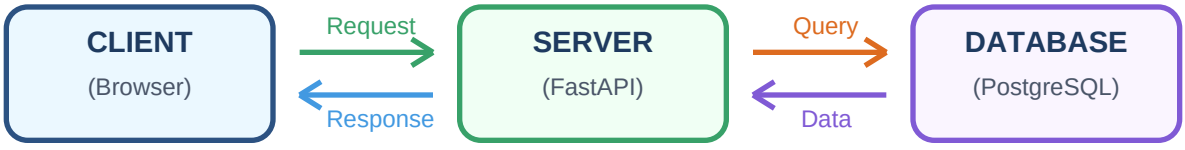
2. The frontend sends a request to the backend via an API.

3. The backend (server, application logic, database) processes the request, fetches or modifies data, and performs calculations.

4. The backend sends the processed data or a response back to the frontend.

5. The frontend displays the result to the user.

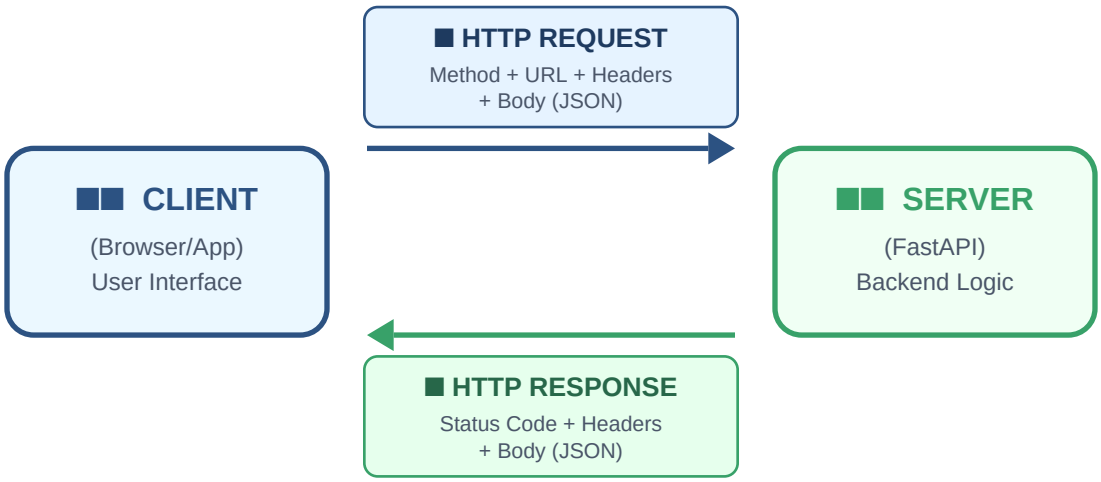
■ Data Flow Architecture



02 HTTP Basics

Status Codes, JSON, and the Request/Response Lifecycle

■ Request/Response Lifecycle



■ HTTP Methods (Verbs)

Method	CRUD	Description	Example
GET	READ	Retrieve data	GET /users
POST	CREATE	Create new resource	POST /users
PUT	UPDATE	Replace entire resource	PUT /users/1
PATCH	UPDATE	Partial update	PATCH /users/1
DELETE	DELETE	Remove resource	DELETE /users/1

■ HTTP Status Codes

Range	Category	Common Codes
2xx	SUCCESS ■	200 OK, 201 Created, 204 No Content
4xx	CLIENT ERROR ■	400 Bad Request, 401 Unauthorized, 404 Not Found
5xx	SERVER ERROR ■	500 Internal Error, 503 Service Unavailable

JSON Format

JSON - The Universal API Language

python

```python
# Python dictionary that becomes JSON
user_data = {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com",
    "is_active": True,
    "scores": [95, 87, 92]
}

# Sent over HTTP as:
# {"id": 1, "name": "John Doe", "email": "john@example.com", ...}
```

03 FastAPI Setup

Installation and Project Structure

■ Installation

Terminal Commands (using UV) bash

```bash
# Initialize project with UV
uv init fastapi_learning
cd fastapi_learning

# Add FastAPI with standard dependencies
uv add fastapi[standard]

# Activate virtual environment
source .venv/bin/activate  # Mac/Linux
.venv\Scripts\activate     # Windows
```

■ Project Structure

Recommended Directory Layout tree

```
fastapi_learning/
■■■ .venv/                # Virtual environment (created by UV)
■■■ app/
■   ■■■ __init__.py
■   ■■■ main.py           # Application entry point
■   ■■■ models.py         # Pydantic models
■   ■■■ routes/
■   ■   ■■■ users.py      # User endpoints
■   ■■■ data/
■       ■■■ store.py      # In-memory storage
■■■ pyproject.toml        # Project config & dependencies
■■■ README.md
```

■ Minimal Application

app/main.py

python

```python
from fastapi import FastAPI

# Create the application instance
app = FastAPI(
    title="Learning API",
    description="A simple API for learning",
    version="1.0.0"
)

# Your first endpoint
@app.get("/")
def read_root():
    return {"message": "Welcome to FastAPI!"}
```

Running the Server

bash

```bash
# Start the development server
uvicorn app.main:app --reload

# Access points:
# API:  http://127.0.0.1:8000
# Docs: http://127.0.0.1:8000/docs
```

FastAPI auto-generates interactive API documentation! Visit /docs for Swagger UI where you can test all endpoints directly in your browser.

04 Creating Endpoints

GET, POST, PUT, DELETE + Path & Query Parameters

Path vs Query Parameters

Aspect	Path Parameters	Query Parameters
Purpose	Identify specific resource	Filter or modify request
Syntax	/users/{user_id}	/users/?skip=0&limit=10
Required?	Usually required	Usually optional
Use Case	Get user #42	Get first 10 active users

Path Parameters

Path Parameter Example python

```python
@app.get("/users/{user_id}")
def get_user(user_id: int):
    """Get a specific user by their ID"""
    # user_id is extracted from the URL
    # GET /users/42 → user_id = 42
    return {"user_id": user_id, "name": "John"}
```

Query Parameters

Query Parameter Example python

```python
@app.get("/users/")
def get_users(
    skip: int = 0,      # Default value = optional
    limit: int = 10,    # Pagination
    active: bool = True # Filter
):
    """Get users with optional filtering"""
    # GET /users/?skip=0&limit=5&active=true
    return {
        "skip": skip,
        "limit": limit,
        "active_only": active
    }
```

■ Combining Both

Combined Parameters python

```python
@app.get("/users/{user_id}/posts")
def get_user_posts(
    user_id: int,           # Path: required
    published: bool = True, # Query: optional
    limit: int = 10         # Query: optional
):
    # GET /users/42/posts?published=true&limit=5
    return {"user_id": user_id, "limit": limit}
```

05 Pydantic Models & Validation

Your Data Bodyguard — Automatic Validation

■ Why Use Pydantic?

Pydantic automatically validates incoming data based on type hints, eliminating the need for manual validation code and preventing bugs from invalid data.

Manual vs Pydantic Validation python

```python
# ■ WITHOUT Pydantic (tedious & error-prone)
@app.post("/users/")
def create_user_bad(data: dict):
    if "name" not in data:
        raise HTTPException(400, "Name required")
    if not isinstance(data.get("age"), int):
        raise HTTPException(400, "Age must be int")
    # ... many more checks needed

# ■ WITH Pydantic (clean & automatic)
class UserCreate(BaseModel):
    name: str = Field(..., min_length=2)
    email: EmailStr
    age: int = Field(..., ge=0, le=120)

@app.post("/users/")
def create_user_good(user: UserCreate):
    return user  # Already validated!
```

■ Complete Model Example

app/models.py

python

```python
from pydantic import BaseModel, EmailStr, Field
from typing import Optional
from datetime import datetime

class UserBase(BaseModel):
    name: str = Field(..., min_length=2, max_length=100)
    email: EmailStr
    age: Optional[int] = Field(None, ge=0, le=150)

class UserCreate(UserBase):
    password: str = Field(..., min_length=8)

class UserUpdate(BaseModel):
    name: Optional[str] = None
    email: Optional[EmailStr] = None

class UserResponse(UserBase):
    id: int
    created_at: datetime
    is_active: bool = True
```

■ Common Field Validators

Validator	Purpose	Example
min_length	Min string length	Field(..., min_length=2)
max_length	Max string length	Field(..., max_length=100)
ge	Greater or equal	Field(..., ge=0)
le	Less or equal	Field(..., le=150)
gt / lt	Greater / Less than	Field(..., gt=0)
regex	Pattern match	Field(..., pattern="^[a-z]+$")

06 Complete CRUD API

Full Working Implementation with In-Memory Storage

■ In-Memory Storage

app/data/store.py

python

```python
# app/data/store.py
users_db: dict = {}
user_id_counter: int = 1

def get_next_id() -> int:
    global user_id_counter
    current = user_id_counter
    user_id_counter += 1
    return current
```

■ CREATE — POST /users/

CREATE Endpoint

python

```python
@app.post("/users/", response_model=UserResponse,
          status_code=status.HTTP_201_CREATED)
def create_user(user: UserCreate):
    # Check for duplicate email
    for existing in users_db.values():
        if existing["email"] == user.email:
            raise HTTPException(400, "Email exists")

    user_id = get_next_id()
    new_user = {
        "id": user_id,
        **user.model_dump(),
        "created_at": datetime.now(),
        "is_active": True
    }
    users_db[user_id] = new_user
    return new_user
```

■ READ — GET /users/ and GET /users/{id}

READ Endpoints

python

```python
@app.get("/users/", response_model=List[UserResponse])
def get_all_users(skip: int = 0, limit: int = 100):
    users = list(users_db.values())
    return users[skip : skip + limit]

@app.get("/users/{user_id}", response_model=UserResponse)
def get_user(user_id: int):
    if user_id not in users_db:
        raise HTTPException(
            status_code=404,
            detail=f"User {user_id} not found"
        )
    return users_db[user_id]
```

✏■ UPDATE — PUT /users/{id}

UPDATE Endpoint

python

```python
@app.put("/users/{user_id}", response_model=UserResponse)
def update_user(user_id: int, user_update: UserUpdate):
    if user_id not in users_db:
        raise HTTPException(404, "User not found")

    stored_user = users_db[user_id]
    update_data = user_update.model_dump(exclude_unset=True)

    for field, value in update_data.items():
        stored_user[field] = value

    return stored_user
```

■■ DELETE — DELETE /users/{id}

DELETE Endpoint

python

```python
@app.delete("/users/{user_id}",
            status_code=status.HTTP_204_NO_CONTENT)
def delete_user(user_id: int):
    if user_id not in users_db:
        raise HTTPException(404, "User not found")

    del users_db[user_id]
    return None  # 204 = No Content
```

07 CORS Configuration

Cross-Origin Resource Sharing Explained

■ What is CORS?

CORS (Cross-Origin Resource Sharing) is a security feature that controls which websites can access your API. By default, browsers block requests from different origins.

Scenario	Origin Match	Default Result
Same server (localhost:3000 → localhost:3000)	Same ✓	Allowed ■
Different port (localhost:3000 → localhost:8000)	Different ✗	Blocked ■
Different domain (myapp.com → api.myapp.com)	Different ✗	Blocked ■

■■ CORS prevents malicious websites from making unauthorized requests using your credentials. Without it, evil-site.com could access your-bank.com's API if you're logged in!

■■ FastAPI CORS Setup

CORS Middleware Configuration

python

```python
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

# Development configuration
app.add_middleware(
    CORSMiddleware,
    allow_origins=[
        "http://localhost:3000",
        "http://127.0.0.1:5500"
    ],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Production Configuration python

```python
# Production: Be specific about allowed origins!
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://myapp.com"],
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT", "DELETE"],
    allow_headers=["Authorization", "Content-Type"],
)
```

■ Never use allow_origins=['*'] with allow_credentials=True in production. This creates a security vulnerability!

08 Testing Your API

Practical Methods to Verify Your Endpoints

Swagger UI (Built-in)

FastAPI automatically generates interactive documentation. Access it at:

- http://127.0.0.1:8000/docs — Interactive Swagger UI for testing endpoints

Using cURL

cURL Commands bash

```bash
# CREATE - Add new user
curl -X POST "http://localhost:8000/users/" \
  -H "Content-Type: application/json" \
  -d '{"name":"John","email":"john@test.com","age":30}'

# READ - Get all users
curl "http://localhost:8000/users/"

# READ - Get specific user
curl "http://localhost:8000/users/1"

# UPDATE - Modify user
curl -X PUT "http://localhost:8000/users/1" \
  -H "Content-Type: application/json" \
  -d '{"name":"John Updated"}'

# DELETE - Remove user
curl -X DELETE "http://localhost:8000/users/1"
```

Using Python Requests

Python Test Script python

```python
import requests

BASE = "http://localhost:8000"

# Create user
resp = requests.post(f"{BASE}/users/", json={
    "name": "Jane",
    "email": "jane@test.com"
})
print(resp.json())

# Get all users
resp = requests.get(f"{BASE}/users/")
print(resp.json())

# Update user
resp = requests.put(f"{BASE}/users/1",
                    json={"name": "Jane Updated"})

# Delete user
resp = requests.delete(f"{BASE}/users/1")
print(resp.status_code)  # 204
```

09 Quick Reference & Summary

Key Concepts Summary for Week 1

■ Key Concepts Summary

Concept	Key Takeaway
Backend	Server-side logic: receives requests, processes data, sends responses
HTTP Methods	GET=read, POST=create, PUT=update, DELETE=remove
Status Codes	2xx=success, 4xx=client error, 5xx=server error
Path Params	/users/{id} — identify specific resource
Query Params	?key=value — filter and paginate
Pydantic	Automatic validation via type hints
CORS	Security: whitelist allowed frontend origins
FastAPI Docs	Auto-generated at /docs and /redoc