FastAPI
Patient Management System

Complete Backend Development Guide

A Comprehensive Guide to Building REST APIs,
HTTP Methods, CRUD Operations, and FastAPI Project Development

Sujil S

`sujil9480@gmail.com`

December 25, 2025

Contents

1 Project Overview

1.1 Introduction

Welcome to the FastAPI Patient Management System project! This comprehensive guide will walk you through building a fully functional REST API from scratch. This project represents a real-world backend application that solves an actual problem faced by healthcare professionals.

1.2 The Problem Statement

1.2.1 Current Situation

When you visit a doctor's clinic, you've probably noticed a common practice:

- Doctors provide prescriptions on clinic letterhead
- They write remarks and medicine names by hand
- Patients are asked to bring these documents on next visit
- Over the years, you accumulate a file of paper prescriptions
- These papers can get misplaced or lost
- Both patient and doctor maintain physical copies

Important Note

The Core Problem: This entire system is offline and prone to data loss. There's no centralized, digital record keeping system. Papers can be lost, making it difficult to track medical history accurately.

1.2.2 Our Solution

We're building a startup that digitizes this entire process:

1. **Digital Platform**: Provide doctors with an app/system
2. **Patient Profiles**: Doctors can maintain profiles for each patient
3. **Centralized Storage**: All patient records stored in one place
4. **Easy Access**: Quick retrieval of patient history
5. **Data Management**: Create, view, update, and delete patient records

1.3 Patient Profile Structure

A typical patient profile in our system contains:

Field	**Description**
Name	Patient's full name
City	Location of patient
Age	Patient's age
Gender	Male/Female/Other
Height	Height in meters
Weight	Weight in kilograms
BMI	Body Mass Index (calculated)

Important Note

While a real-world medical system would have much more information (medical history, allergies, current medications, etc.), we're keeping it simple for this learning project. However, the same principles can be extended to include any amount of data.

1.4 What We're Building

1.4.1 Our Task as Backend Engineers

Important Distinction

We are NOT building the actual app (that's frontend work). We're building the **API** - the backend system that the app will communicate with.

The API will provide approximately 5 endpoints:

1. **Create Patient**: Add new patient records
2. **View All Patients**: Retrieve all patient records
3. **View Specific Patient**: Get details of one patient by ID
4. **Update Patient**: Modify existing patient information
5. **Delete Patient**: Remove patient from database

1.5 Data Storage Approach

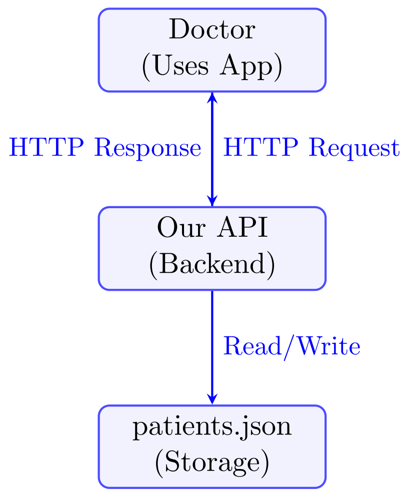
1.5.1 Development vs Production

Ideal Approach: Store data in a proper database (PostgreSQL, MongoDB, etc.)
Our Approach: For this learning project, we'll use a JSON file

Important Note

Using a JSON file instead of a database doesn't change the fundamental logic. The methods and operations remain the same. This approach helps us focus on API development without the complexity of database setup.

1.6 API Functionality Overview



1.6.1 The Five Core Operations

1. Create New Patient

- Doctor fills form in app
- Data sent to API endpoint
- API adds patient to JSON file

2. View All Patients

- Request sent to view endpoint
- API retrieves all records from JSON
- Returns complete patient list

3. View Specific Patient

- Request with patient ID (e.g., Patient_1, Patient_5, Patient_10)
- API finds that specific record
- Returns individual patient profile

4. Update Patient Record

- Doctor modifies patient information
- Example: Weight changed after 1 year
- API updates the record and recalculates BMI

5. Delete Patient

- Request with patient ID
- API removes patient from database
- Permanent deletion

2 Understanding HTTP Methods

Before we start building the API, we need to understand HTTP methods thoroughly. This concept will appear throughout the entire project.

2.1 Software Classification by User Interaction

Software can be classified into two categories based on user interaction:

Static Software	**Dynamic Software**
Minimal user interaction	High user interaction
One-way communication	Two-way communication
Used for receiving information	Used for active data manipulation
Examples: Calendar, Clock	Examples: Microsoft Excel, Word, Instagram

2.2 Examples of Static Software

2.2.1 Calendar Application

- Click to open calendar
- Shows today's date
- Can view future dates
- **No real interaction** - just viewing information
- One-way communication: Software tells you information

2.2.2 Clock Application

- Used only to check time
- No user interaction
- Pure information display

2.3 Examples of Dynamic Software

2.3.1 Microsoft Excel

Think about how many ways you interact with Excel:

- Create new worksheets
- Enter data in cells
- Perform analysis on existing data
- Modify existing data
- Delete data
- Create formulas and charts

The level of interaction is extremely high!

2.3.2 Microsoft Word

- Create new documents
- Read existing documents
- Update existing documents
- Delete documents

2.4 The CRUD Paradigm

Fundamental Concept

Question: How many types of interactions can you have with ANY dynamic software?
Answer: Only FOUR types of interactions, known as **CRUD**

2.4.1 What is CRUD?

Letter	Operation	Description
C	**Create**	Adding new data/records
R	**Retrieve/Read**	Viewing/reading existing data
U	**Update**	Modifying existing data
D	**Delete**	Removing data

2.5 CRUD in Microsoft Excel

Let's verify this with Excel:

1. **Create**: Creating cells and entering data
2. **Retrieve**: Viewing existing cells and data
3. **Update**: Going to existing cells and making changes
4. **Delete**: Deleting existing cells

Conclusion: Every interaction falls into one of these four categories!

2.6 CRUD in Microsoft Word

1. **Create**: Creating new documents
2. **Retrieve**: Reading existing documents
3. **Update**: Modifying existing documents
4. **Delete**: Deleting existing documents

Important Note

Universal Truth: Any dynamic software, regardless of complexity, only allows these four types of interactions. Think of any software - you'll realize all interactions fall into CRUD operations.

2.7 From Software to Websites

2.7.1 What is a Website?

Important Clarification

Websites are not fundamentally different from software. They are software with one key distinction:

- **Normal Software**: Installed and used on the SAME machine
- **Website**: Installed on one machine (server), used from another machine (client)

2.7.2 Client-Server Architecture

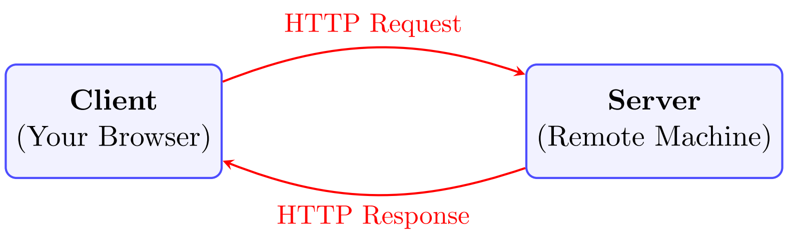
Example: Microsoft Excel

- Installed on YOUR machine
- Used on YOUR machine
- Local operation

Example: Website

- Installed on a **server** (remote machine)
- Accessed from a **client** (your machine)
- Communication over the internet

2.8 Client-Server Communication



- **Client**: Requests something from server
- **Server**: Responds to client's request
- **Protocol**: Communication happens via HTTP (Hypertext Transfer Protocol)

2.9 Website Classification

Just like software, websites are also classified:

2.9.1 Static Websites

- Minimal user interaction
- Primarily for displaying information
- Examples: Blogs, government websites
- One-way information flow

2.9.2 Dynamic Websites

- High user interaction
- Active data manipulation
- Examples: Instagram, Facebook, Zomato
- Two-way communication

2.10 CRUD in Dynamic Websites

Important Note

Key Point: The same CRUD principle applies to websites! No matter how many interactions a website allows, they all fall into these four categories.

2.10.1 Instagram Example

Let's analyze Instagram interactions:

Create Operations:

- Upload new photo/video posts
- Write new comments
- Create stories
- Send new messages

Retrieve Operations:

- Scroll through feed
- View someone's profile
- Read comments
- View stories

Update Operations:

- Edit your profile
- Edit your comments
- Update bio

Delete Operations:

- Delete your posts
- Delete your comments
- Remove followers

2.10.2 Zomato Example

Create: Place a new order **Retrieve**: View past orders **Update**: Update delivery address **Delete**: Delete saved address

3 HTTP Methods (Verbs)

3.1 The Communication Challenge

Since dynamic websites communicate over the internet (client ↔ server via HTTP), there's an important requirement:

Warning

Critical Requirement: When performing any CRUD operation, the HTTP request MUST specify which type of interaction you want to perform!

Why?

- HTTP requests travel over the internet
- Server needs to understand what the client wants
- Different operations require different handling
- Ambiguity would cause errors

3.2 HTTP Verbs/Methods

To specify the type of interaction, we add a **verb** to the HTTP request. These verbs are called **HTTP Methods**.

CRUD	HTTP Method	Purpose
Create	**POST**	Send data to server to create new resource
Retrieve	**GET**	Request data from server
Update	**PUT/PATCH**	Modify existing resource on server
Delete	**DELETE**	Remove resource from server

3.3 The GET Method

Purpose: Retrieve/Read data from server

When to use:

- Viewing a profile page
- Scrolling through feed
- Reading comments
- Accessing any page

GET Request Example

Scenario: You want to view your profile page on a website

What happens:

1. Your browser sends HTTP request with GET method
2. Request includes: ”I want to retrieve my profile page”
3. Server understands this is a retrieve operation
4. Server sends back the profile page data

Request format:

```
GET /profile HTTP/1.1
Host: example.com
```

3.4 The POST Method

Purpose: Send data to server to create new resources

When to use:

- Submitting login form
- Submitting registration form
- Uploading a new post
- Sending any form data

POST Request Example

Scenario: You're filling a login form

What happens:

1. You enter email and password
2. Click "Login" button
3. Browser sends HTTP request with POST method
4. Request body contains your credentials
5. Server receives and processes the data

Request format:

```
POST /authenticate HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "email": "user@example.com",
    "password": "password123"
}
```

3.5 The PUT Method

Purpose: Update/Modify existing resource

When to use:

- Editing profile information
- Updating a post
- Modifying settings
- Changing existing data

PUT Request Example

Scenario: Updating your profile bio

What happens:

1. You edit your bio
2. Click "Save"
3. Browser sends HTTP request with PUT method
4. Server updates your profile

3.6 The DELETE Method

Purpose: Remove resource from server

When to use:

- Deleting a post
- Removing a comment
- Deleting an account
- Removing any data

3.7 HTTP Methods Summary

Key Concept

For EVERY type of interaction in the HTTP body, you send a verb. These verbs collectively are called **HTTP Methods**.

Most commonly used:

- **GET and POST**: Used most frequently
- **PUT and DELETE**: Used less frequently but still important

Important Note

These four HTTP methods are essential knowledge for anyone building websites or APIs. They form the foundation of REST API design.

4 HTTP Methods in Action

Let's see HTTP methods in real-world usage by inspecting browser network traffic.

4.1 Inspecting Network Requests

4.1.1 Setup

To view HTTP methods in your browser:

1. Open any website
2. Right-click and select "Inspect" (or press F12)
3. Go to the "Network" tab
4. Perform actions on the website
5. Watch requests appear in the network tab

4.2 Example 1: GET Request

4.2.1 Scenario: Accessing a Courses Page

Action: Click on "Courses" link

Operation Type: Retrieve/Read (viewing a page)

Expected Method: GET

Network Tab Observation

```
Request URL: https://example.com/store
Request Method: GET
Status Code: 200 OK
```

What we see:

- Request URL shows the page we're accessing
- Request Method is GET (as expected)
- This confirms: viewing a page uses GET

4.3 Example 2: POST Request

4.3.1 Scenario: Logging In

Action: Fill login form and click "Login"

Operation Type: Create (sending data to server)

Expected Method: POST

Login Form Submission

Steps:

1. Click "Login" button
2. Click "Continue with Email"

3. Enter email: dummy@gmail.com
4. Enter password: 1234 (intentionally wrong)
5. Click ”Next”

Network Tab Observation:

```
Request URL: https://example.com/authenticate
Request Method: POST
Status Code: 401 Unauthorized
```

What we see:

- Request URL: /authenticate endpoint
- Request Method: POST (as expected)
- This confirms: sending form data uses POST
- Status 401: Authentication failed (wrong password)

4.4 Key Observations

Understanding HTTP Methods in Practice

GET Requests:

- Triggered when viewing pages
- No data sent in body
- Just retrieving information

POST Requests:

- Triggered when submitting forms
- Data sent in request body
- Creating or sending information

5 Starting the Project

Now that we understand HTTP methods, let's start building our Patient Management API!

5.1 Environment Setup

5.1.1 Activating Virtual Environment

```
# Windows
env\Scripts\activate.bat

# Linux/Mac
source env/bin/activate
```

5.2 Creating the Data File

5.2.1 patients.json Structure

Create a file named `patients.json` with sample data:

```json
{
  "P001": {
    "name": "Ananya Verma",
    "city": "Guwahati",
    "age": 28,
    "gender": "female",
    "height": 1.65,
    "weight": 90.0,
    "bmi": 33.06,
    "verdict": "Obese"
  },
  "P002": {
    "name": "Ravi Mehta",
    "city": "Mumbai",
    "age": 35,
    "gender": "male",
    "height": 1.75,
    "weight": 85,
    "bmi": 27.76,
    "verdict": "Overweight"
  },
  "P003": {
    "name": "Sneha Kulkarni",
    "city": "Pune",
    "age": 22,
    "gender": "female",
    "height": 1.6,
    "weight": 45,
    "bmi": 17.58,
    "verdict": "Underweight"
  },
  "P004": {
    "name": "Arjun Verma",
    "city": "Mumbai",
    "age": 40,
    "gender": "male",
    "height": 1.8,
    "weight": 90.0,
    "bmi": 27.78,
    "verdict": "Normal"
  },
  "P005": {
    "name": "Neha Sinha",
    "city": "Kolkata",
    "age": 30,
    "gender": "female",
    "height": 1.55,
    "weight": 75,
    "bmi": 31.22,
    "verdict": "Obese"
  }
}
```

Important Note

This JSON file acts as our ”database” for this learning project. Each patient has a unique ID (P001, P002, etc.) and stores all relevant information. As new patients are added, they'll be appended to this file.

5.3 Understanding the Data Structure

Each patient record contains:

- **Patient ID**: Unique identifier (P001, P002, etc.)
- **name**: Patient's full name
- **city**: Location
- **age**: Patient's age in years
- **gender**: Male/Female
- **height**: Height in meters
- **weight**: Weight in kilograms
- **bmi**: Body Mass Index (calculated)
- **verdict**: Health status based on BMI

6 Building the First API Endpoint

6.1 Modifying Existing Code

Let's update our basic FastAPI app from the previous session.

6.1.1 Updated main.py - Initial Version

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def hello():
    return {"message": "Hello World"}

@app.get("/about")
def about():
    return {"message": "This is the about page"}
```

6.1.2 Updated main.py - Modified Version

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def hello():
    return {"message": "Patient Management System API"}

@app.get("/about")
def about():
    return {"message": "A fully functional API to manage patient
   records"}
```

6.2 Creating a Helper Function

Before building endpoints, we need a reusable function to load data from the JSON file.

6.2.1 Why We Need a Helper Function

- We'll need to read from patients.json repeatedly
- Multiple endpoints will require patient data
- Better to write once and reuse
- Follows DRY principle (Don't Repeat Yourself)

6.2.2 The load_data() Function

```python
import json

def load_data():
    """
    Helper function to load patient data from JSON file
    Returns: Dictionary containing all patient records
    """
    with open("patients.json", "r") as f:
        data = json.load(f)

    return data
```

How it works:

1. Opens the patients.json file in read mode
2. Uses json.load() to parse the JSON data
3. Returns the data as a Python dictionary
4. Can be called anytime we need patient data

6.3 Building the View All Patients Endpoint

6.3.1 Endpoint Specification

Endpoint Details

Route: /view
HTTP Method: GET (retrieving data)
Purpose: Return all patient records
Response: Complete JSON data from patients.json

6.3.2 Complete Code

```python
from fastapi import FastAPI
import json

app = FastAPI()

def load_data():
    """Load patient data from JSON file"""
    with open("patients.json", "r") as f:
        data = json.load(f)
    return data

@app.get("/")
def hello():
    return {"message": "Patient Management System API"}

@app.get("/about")
def about():
    return {"message": "A fully functional API to manage patient
    records"}

@app.get("/view")
def view():
    """
    Endpoint to retrieve all patient records
    Returns: All patient data from JSON file
    """
    data = load_data()
    return data
```

6.4 Code Explanation

6.4.1 Import Statements

```
from fastapi import FastAPI
import json
```

- `FastAPI`: Core framework for building the API
- `json`: Module for parsing JSON files

6.4.2 The View Endpoint

```
@app.get("/view")
def view():
    data = load_data()
    return data
```

Step-by-step execution:

1. Decorator `@app.get("/view")` defines route and method
2. Function `view()` handles requests to this endpoint
3. Calls `load_data()` to fetch all patient records
4. Returns data as JSON response automatically

Important Note

FastAPI automatically converts Python dictionaries to JSON format in the HTTP response. You don't need to manually serialize the data!

7 Testing the API

7.1 Starting the Server

7.1.1 Using Uvicorn

```
uvicorn main:app --reload
```

Command breakdown:

- `uvicorn`: ASGI server for running FastAPI
- `main`: Name of Python file (main.py)
- `app`: Name of FastAPI instance
- `--reload`: Auto-reload on code changes (development only)

7.1.2 Server Output

```
INFO:     Uvicorn running on http://127.0.0.1:8000
INFO:     Application startup complete.
```

Your API is now running at: `http://127.0.0.1:8000`

7.2 Testing Endpoints in Browser

7.2.1 Home Endpoint

URL: `http://127.0.0.1:8000/`

Response:

```
{
    "message": "Patient Management System API"
}
```

7.2.2 About Endpoint

URL: `http://127.0.0.1:8000/about`

Response:

```
{
    "message": "A fully functional API to manage patient records"
}
```

7.2.3 View All Patients Endpoint

URL: `http://127.0.0.1:8000/view`

Response: (Complete patient data)

```
{
  "P001": {
    "name": "Ananya Verma",
    "city": "Guwahati",
    "age": 28,
    "gender": "female",
    "height": 1.65,
    "weight": 90.0,
    "bmi": 33.06,
    "verdict": "Obese"
  },
  "P002": {
    "name": "Ravi Mehta",
    "city": "Mumbai",
    "age": 35,
    "gender": "male",
    "height": 1.75,
    "weight": 85,
    "bmi": 27.76,
    "verdict": "Overweight"
  },
  // ... rest of the patient data
}
```

7.3 Using FastAPI's Interactive Documentation

7.3.1 Accessing Swagger UI

FastAPI automatically generates interactive API documentation!

URL: `http://127.0.0.1:8000/docs`

Automatic Documentation

FastAPI uses OpenAPI standards to create beautiful, interactive documentation. This is one of FastAPI's killer features - you get professional API docs without writing a single line of documentation code!

7.3.2 Testing in Swagger UI

1. Navigate to `http://127.0.0.1:8000/docs`
2. You'll see all available endpoints listed
3. Find the `GET /view` endpoint
4. Click to expand it
5. Click "Try it out" button
6. Click "Execute" button
7. View the response below

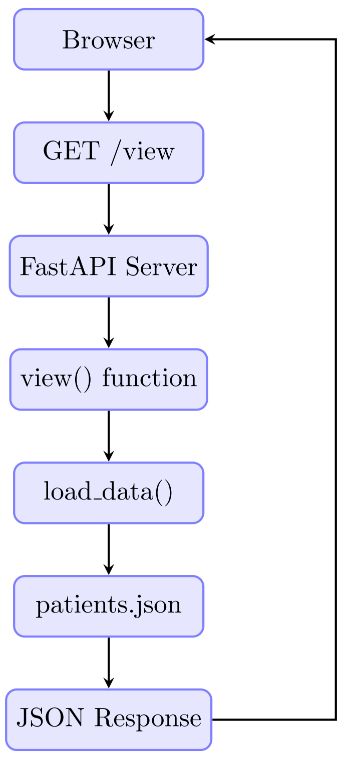
Swagger UI Response

Response Code: 200 OK
Response Body: Shows all patient data
Response Headers: Content-Type: application/json

This confirms our endpoint is working perfectly!

7.4 Understanding the Response

7.4.1 What Happens When You Access /view


$$\begin{tikzcd}
\text{Browser} \arrow[d] \\
\text{GET /view} \arrow[d] \\
\text{FastAPI Server} \arrow[d] \\
\text{view() function} \arrow[d] \\
\text{load\_data()} \arrow[d] \\
\text{patients.json} \arrow[d] \\
\text{JSON Response} \arrow[uuuuuu, bend right]
\end{tikzcd}$$

Step-by-step flow:

1. Browser sends GET request to /view
2. FastAPI routes request to view() function
3. view() calls load_data() helper function
4. load_data() opens and reads patients.json
5. Data returned to view() function
6. view() returns data to FastAPI
7. FastAPI serializes to JSON and sends response
8. Browser receives and displays the data

8 Complete Code Reference

8.1 Final main.py

```python
from fastapi import FastAPI
import json

app = FastAPI()

def load_data():
    """
    Helper function to load patient data from JSON file.

    Returns:
        dict: Dictionary containing all patient records
    """
    with open("patients.json", "r") as f:
        data = json.load(f)
    return data

@app.get("/")
def hello():
    """
    Home endpoint - API welcome message.

    Returns:
        dict: Welcome message
    """
    return {"message": "Patient Management System API"}

@app.get("/about")
def about():
    """
    About endpoint - API description.

    Returns:
        dict: API description message
    """
    return {"message": "A fully functional API to manage patient
   records"}

@app.get("/view")
def view():
    """
    View all patients endpoint.
    Retrieves and returns all patient records from the database.

    Returns:
        dict: Complete patient data with all records
    """
    data = load_data()
    return data
```

8.2 patients.json Structure

```json
{
  "P001": {
    "name": "Ananya Verma",
    "city": "Guwahati",
    "age": 28,
    "gender": "female",
    "height": 1.65,
    "weight": 90.0,
    "bmi": 33.06,
    "verdict": "Obese"
  },
  "P002": {
    "name": "Ravi Mehta",
    "city": "Mumbai",
    "age": 35,
    "gender": "male",
    "height": 1.75,
    "weight": 85,
    "bmi": 27.76,
    "verdict": "Overweight"
  },
  "P003": {
    "name": "Sneha Kulkarni",
    "city": "Pune",
    "age": 22,
    "gender": "female",
    "height": 1.6,
    "weight": 45,
    "bmi": 17.58,
    "verdict": "Underweight"
  },
  "P004": {
    "name": "Arjun Verma",
    "city": "Mumbai",
    "age": 40,
    "gender": "male",
    "height": 1.8,
    "weight": 90.0,
    "bmi": 27.78,
    "verdict": "Normal"
  },
  "P005": {
    "name": "Neha Sinha",
    "city": "Kolkata",
    "age": 30,
    "gender": "female",
    "height": 1.55,
    "weight": 75,
    "bmi": 31.22,
    "verdict": "Obese"
  }
}
```

8.3 Project Structure

```
patient-management-api/
|-- env/                        # Virtual environment
|-- main.py                     # FastAPI application
|-- patients.json               # Patient database (JSON)
|-- requirements.txt            # Python dependencies
```

8.4 requirements.txt

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
```

9 Key Concepts Summary

9.1 What We Learned

9.1.1 1. Project Understanding

- Real-world problem: Digitizing patient records
- Solution: REST API for patient management
- Backend focus: We build the API, not the app
- Storage: JSON file (learning project) vs Database (production)

9.1.2 2. Software Classification

Static Software	**Dynamic Software**
Minimal interaction	High interaction
One-way communication	Two-way communication
Examples: Calendar, Clock	Examples: Excel, Instagram

9.1.3 3. CRUD Operations

Universal Pattern

ALL dynamic software/websites support only 4 types of interactions:

- **C**reate: Adding new data
- **R**etrieve: Reading existing data
- **U**pdate: Modifying existing data
- **D**elete: Removing data

9.1.4 4. HTTP Methods

CRUD	**HTTP Method**	**Usage**
Create	POST	Submit forms, create resources
Retrieve	GET	View pages, fetch data
Update	PUT/PATCH	Modify existing resources
Delete	DELETE	Remove resources

9.1.5 5. Client-Server Architecture

- **Client**: User's browser/device (sends requests)
- **Server**: Remote machine hosting the API (sends responses)
- **Protocol**: HTTP (communication standard)
- **Request**: Contains method, URL, headers, body
- **Response**: Contains status code, headers, body

9.1.6 6. FastAPI Basics

- Framework for building APIs in Python
- Automatic JSON serialization
- Built-in interactive documentation
- Type hints and data validation
- Fast performance (async support)

9.2 Practical Implementation

9.2.1 What We Built

1. **Home Endpoint** (/): Welcome message
2. **About Endpoint** (/about): API description
3. **View All Endpoint** (/view): Get all patient records

9.2.2 Code Structure

```python
# 1. Import necessary modules
from fastapi import FastAPI
import json

# 2. Create FastAPI instance
app = FastAPI()

# 3. Helper function
def load_data():
    # Load data from JSON file
    pass

# 4. Define endpoints
@app.get("/route")
def function_name():
    # Endpoint logic
    return response_data
```

9.3 Important Principles

Best Practices

1. **DRY Principle**: Don't Repeat Yourself (use helper functions)
2. **RESTful Design**: Follow REST conventions for API design
3. **Clear Naming**: Use descriptive function and variable names
4. **Proper HTTP Methods**: Use correct method for each operation
5. **Documentation**: Write docstrings for functions
6. **Error Handling**: Plan for failure cases
7. **Consistent Response Format**: Standardize API responses

10 Common Issues and Solutions

10.1 Server Won't Start

10.1.1 Problem: ModuleNotFoundError

```
ModuleNotFoundError: No module named 'fastapi'
```

Solution:

```
# Ensure virtual environment is activated
# Then install FastAPI
pip install fastapi uvicorn
```

10.1.2 Problem: Port Already in Use

```
ERROR: [Errno 48] Address already in use
```

Solution:

```
# Use different port
uvicorn main:app --reload --port 8001

# Or kill existing process on port 8000
# Windows:
netstat -ano | findstr :8000
taskkill /PID <PID> /F

# Linux/Mac:
lsof -ti:8000 | xargs kill
```

10.2 JSON File Issues

10.2.1 Problem: FileNotFoundError

```
FileNotFoundError: [Errno 2] No such file or directory: 'patients.json'
```

Solution:

- Ensure patients.json is in the same directory as main.py
- Check file name spelling (case-sensitive)
- Verify file path in load_data() function

10.2.2 Problem: JSON Decode Error

```
json.decoder.JSONDecodeError: Expecting property name enclosed in double quotes
```

Solution:

- Validate JSON syntax at jsonlint.com
- Ensure all keys are in double quotes
- Check for trailing commas
- Verify proper bracket/brace closure

10.3 Browser Issues

10.3.1 Problem: 404 Not Found

Causes and Solutions:

- Wrong URL: Check spelling of endpoint
- Server not running: Ensure uvicorn is active
- Wrong port: Verify you're using correct port number

10.3.2 Problem: Empty Response

Solutions:

- Check if patients.json has data
- Verify load_data() returns data correctly
- Check browser console for errors

11 Quick Reference Guide

11.1 Essential Commands

Virtual Environment

```
# Create virtual environment
python -m venv env

# Activate (Windows)
env\Scripts\activate.bat

# Activate (Linux/Mac)
source env/bin/activate

# Deactivate
deactivate
```

Installation

```
# Install FastAPI and Uvicorn
pip install fastapi uvicorn[standard]

# Install from requirements.txt
pip install -r requirements.txt

# Create requirements.txt
pip freeze > requirements.txt
```

Running Server

```
# Basic command
uvicorn main:app

# With auto-reload (development)
uvicorn main:app --reload

# Custom port
uvicorn main:app --reload --port 8001

# Custom host (accessible from network)
uvicorn main:app --host 0.0.0.0 --port 8000
```

11.2 FastAPI Endpoint Template

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/endpoint")              # HTTP Method and Route
def function_name():               # Function name
    """
    Endpoint description
    """
    # Your logic here
    return {"key": "value"}      # Return dictionary (becomes JSON)
```

11.3 HTTP Methods Reference

Method	**Decorator**	**Purpose**
GET	@app.get()	Retrieve data, read operations
POST	@app.post()	Create new resources, submit data
PUT	@app.put()	Update entire resource
PATCH	@app.patch()	Partial update of resource
DELETE	@app.delete()	Delete resource

11.4 Common HTTP Status Codes

Code	**Status**	**Meaning**
200	OK	Request succeeded
201	Created	Resource created successfully
400	Bad Request	Invalid request data
401	Unauthorized	Authentication required
404	Not Found	Resource not found
500	Internal Server Error	Server error

End of FastAPI Patient Management System Guide - Session 1

"The only way to learn a new programming language is by writing programs in it."
— Dennis Ritchie

Keep coding, keep learning!
See you in the next session where we'll implement more CRUD operations!