FastAPI

Complete Reference Guide

Modern API Development with Python

A Comprehensive Guide to Building
High-Performance APIs with FastAPI

Learning Notes

Based on Industry Best Practices

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to FastAPI **2**
1.1 What is FastAPI? . . . 2
1.2 Why FastAPI for Data Science and AI/ML? . . . 2
1.3 The Foundation: Built on Two Libraries . . . 2
1.3.1 1. Starlette . . . 2
1.3.2 2. Pydantic . . . 3
1.4 Architecture Overview . . . 3

2 Core Philosophy of FastAPI **4**
2.1 The Two Primary Objectives . . . 4
2.2 Philosophy 1: Fast to Run . . . 4
2.3 Philosophy 2: Fast to Code . . . 4

3 Understanding API Architecture **5**
3.1 API Components . . . 5
3.2 Complete Request-Response Flow . . . 5
3.2.1 Step-by-Step Flow . . . 5
3.3 Visual Flow Diagram . . . 6

4 Flask vs FastAPI: Architecture Comparison **7**
4.1 Flask Architecture . . . 7
4.1.1 Flask Components . . . 7
4.1.2 WSGI (Web Server Gateway Interface) . . . 7
4.1.3 Example: Multiple Clients . . . 7
4.1.4 Werkzeug . . . 8
4.1.5 Gunicorn Web Server . . . 8
4.1.6 Flask API Code . . . 8
4.2 Flask Architecture Summary . . . 9
4.3 FastAPI Architecture . . . 10
4.3.1 FastAPI Components . . . 10
4.3.2 ASGI (Asynchronous Server Gateway Interface) . . . 10
4.3.3 Starlette . . . 10
4.3.4 Uvicorn Web Server . . . 11
4.3.5 Async/Await in Python . . . 11
4.4 FastAPI Architecture Summary . . . 12
4.5 Side-by-Side Comparison . . . 12
4.6 The Restaurant Analogy . . . 13

5 Why FastAPI is Fast to Code **14**
5.1 Three Key Aspects . . . 14
5.2 1. Automatic Input Validation . . . 14
5.2.1 The Problem in Python . . . 14
5.2.2 Why This Matters for APIs . . . 14
5.2.3 FastAPI + Pydantic Solution . . . 14
5.3 2. Auto-Generated Interactive Documentation . . . 15
5.3.1 The Documentation Challenge . . . 15
5.3.2 FastAPI's Solution . . . 16
5.4 3. Seamless Modern Library Integration . . . 16
5.4.1 Machine Learning Libraries . . . 16

5.4.2 Authentication . . . 16
5.4.3 Database Integration . . . 17
5.4.4 Deployment . . . 17
5.5 Summary: Fast to Code . . . 17

6 Installing and Setting Up FastAPI 18
6.1 Prerequisites . . . 18
6.2 Step-by-Step Setup . . . 18
6.2.1 Step 1: Create Project Folder . . . 18
6.2.2 Step 2: Open in VS Code . . . 18
6.2.3 Step 3: Create Virtual Environment . . . 18
6.2.4 Step 4: Activate Virtual Environment . . . 18
6.2.5 Step 5: Install Required Packages . . . 19
6.3 Verifying Installation . . . 19

7 Building Your First FastAPI Application 20
7.1 Hello World API . . . 20
7.1.1 Step 1: Create Python File . . . 20
7.1.2 Step 2: Write the Code . . . 20
7.2 Code Explanation . . . 20
7.2.1 1. Import FastAPI . . . 20
7.2.2 2. Create App Instance . . . 20
7.2.3 3. Define Route . . . 20
7.2.4 4. Define Handler Function . . . 21
7.3 Running the API . . . 21
7.3.1 Start the Server . . . 21
7.3.2 Expected Output . . . 21
7.3.3 Test the API . . . 22
7.4 Adding More Endpoints . . . 22
7.4.1 Updated Code . . . 22
7.4.2 What Changed? . . . 22
7.4.3 Auto-Reload Magic . . . 22
7.4.4 Testing Multiple Endpoints . . . 23
7.5 Understanding Routes . . . 23

8 Interactive API Documentation 24
8.1 Accessing Auto-Generated Docs . . . 24
8.1.1 Swagger UI Documentation . . . 24
8.2 What You'll See . . . 24
8.3 Example Documentation View . . . 24
8.4 Interactive Testing . . . 24
8.4.1 Step 1: Expand Endpoint . . . 24
8.4.2 Step 2: Click "Try it out" . . . 24
8.4.3 Step 3: Execute . . . 24
8.4.4 Step 4: View Response . . . 24
8.5 Benefits of Auto-Documentation . . . 25
8.6 Alternative Documentation . . . 25

9 FastAPI Core Concepts **26**
9.1 Understanding Decorators . . . 26
9.1.1 What is a Decorator? . . . 26
9.1.2 What @app.get() Does . . . 26
9.2 HTTP Methods in FastAPI . . . 26
9.2.1 Common HTTP Methods . . . 26
9.2.2 GET vs POST . . . 26
9.3 Return Values . . . 27
9.3.1 Automatic JSON Conversion . . . 27
9.3.2 What You Can Return . . . 27
9.4 Path Parameters . . . 27
9.5 FastAPI Application Object . . . 27
9.5.1 The app Instance . . . 27
9.5.2 Customizing the App . . . 28

10 Project Organization and Best Practices **29**
10.1 Recommended Project Structure . . . 29
10.2 Creating requirements.txt . . . 29
10.3 Development Best Practices . . . 29
10.4 Common Commands Reference . . . 30

1 Introduction to FastAPI

1.1 What is FastAPI?

Definition

FastAPI is a modern, high-performance web framework for building APIs with Python.

FastAPI is a Python framework that enables developers to build high-performance, industry-grade APIs efficiently. It captures the essence of modern API development by combining speed, ease of use, and robust features.

1.2 Why FastAPI for Data Science and AI/ML?

FastAPI has become essential for professionals working in:

- **Data Science**: Deploy data analysis pipelines as APIs
- **Machine Learning**: Serve ML models in production
- **Artificial Intelligence**: Create AI-powered applications
- **Deep Learning**: Deploy neural networks as web services

Important Note

In modern ML/AI workflows, building an API is often the final step to make your models accessible to applications, allowing them to serve predictions in real-time.

1.3 The Foundation: Built on Two Libraries

FastAPI is built on top of two popular Python libraries:

1.3.1 1. Starlette

Starlette is responsible for handling HTTP communication in FastAPI.

- **Role**: Manages how your API receives requests and sends back responses
- **Functionality**: Handles all HTTP request/response processing
- **Performance**: Provides asynchronous request handling

What it does:

- Receives HTTP requests from clients
- Processes the request data
- Sends HTTP responses back to clients
- Manages the web server interface

1.3.2 2. Pydantic

Pydantic is a data validation library used in FastAPI.

- **Role**: Validates incoming data format and types
- **Purpose**: Ensures data integrity and type safety
- **Benefit**: Automatic data validation without manual checks

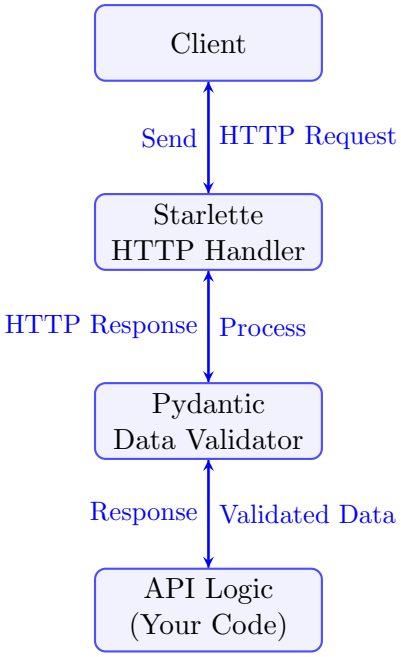
Why Pydantic is Important in APIs

Consider an API that takes two station names and a date to return train information:
Without Pydantic: You must manually check:

- Are station names in string format?
- Are they from the valid list of stations?
- Is the date in correct format?

With Pydantic: All these checks happen automatically behind the scenes!

1.4 Architecture Overview



2 Core Philosophy of FastAPI

2.1 The Two Primary Objectives

FastAPI was created to solve two major problems that existed with older frameworks like Flask:

1. **Performance Issues**: Older frameworks had slow response times and latency issues
2. **Development Complexity**: Writing APIs required extensive boilerplate code

2.2 Philosophy 1: Fast to Run

Fast to Run

FastAPI-built APIs are **fast in execution**, meaning they:

- Handle requests very quickly
- Support concurrent users efficiently
- Have minimal latency
- Perform well under load

2.3 Philosophy 2: Fast to Code

Fast to Code

FastAPI makes API development **fast to write**, meaning:

- Less boilerplate code required
- Fewer lines of code needed
- Quick development cycle
- Clean, readable syntax

Important Note

The name "FastAPI" reflects both philosophies:

- **Fast** = Fast to run (performance)
- **Fast** = Fast to code (development speed)

3 Understanding API Architecture

3.1 API Components

Every API consists of two essential components:

1. **API Code**: The business logic that processes requests
2. **Web Server**: The component that listens for HTTP requests

3.2 Complete Request-Response Flow

Let's understand the complete flow using a machine learning prediction API example:

Example Scenario

API Endpoint: `/predict`
Inputs:

- Feature1: some value
- Feature2: some value

Output: Model prediction

3.2.1 Step-by-Step Flow

Step 1: Client Makes Request

- User sends feature values to the API endpoint
- Browser/software converts this into an HTTP request

Step 2: HTTP Request Creation

- Request method (POST, GET, etc.)
- Endpoint URL (`/predict`)
- Headers (Content-Type, Content-Length, etc.)
- Body (Feature values)
- Host URL

Step 3: Web Server Receives Request

- Web server listens on ports continuously
- Captures incoming HTTP request
- Passes it to the gateway interface

Step 4: Server Gateway Interface (SGI)

- Acts as a translator between HTTP and Python
- Converts HTTP format to Python-understandable format
- Enables two-way communication

Warning

The Problem: Python cannot directly understand HTTP requests!

The Solution: Server Gateway Interface (SGI) translates between HTTP and Python formats.

Step 5: API Code Execution

- Python code receives feature values
- Loads ML model
- Calls prediction function
- Generates result

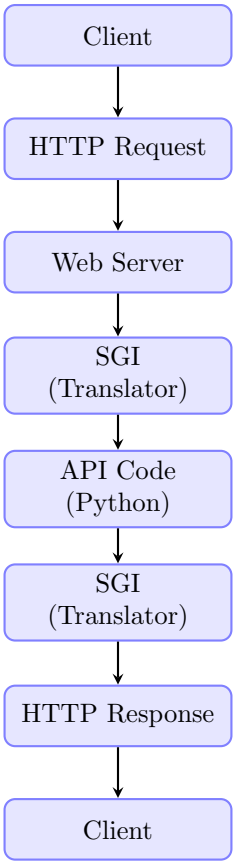
Step 6: Response Generation

- Python output converted back to HTTP format by SGI
- Status code added (e.g., 200 for success)
- Response headers added
- Complete HTTP response created

Step 7: Response Sent to Client

- Web server sends HTTP response
- Client receives prediction result

3.3 Visual Flow Diagram



4 Flask vs FastAPI: Architecture Comparison

4.1 Flask Architecture

Flask is a traditional, well-established Python web framework. Let's examine its components:

4.1.1 Flask Components

1. **SGI Protocol**: WSGI (Web Server Gateway Interface)
2. **WSGI Implementation**: Werkzeug library
3. **Web Server**: Gunicorn
4. **API Code**: Synchronous Python code

4.1.2 WSGI (Web Server Gateway Interface)

What is WSGI?

WSGI is a specification that standardizes how web servers and Python web applications/frameworks communicate.

Key Characteristics:

- Older, mature protocol
- **Synchronous** in nature
- Processes one request at a time
- **Blocking architecture**

Warning

The Biggest Problem with WSGI:
WSGI follows a **synchronous, blocking architecture**.

What this means:

- Can process only ONE request at a time
- When processing one request, others must wait
- Resources are blocked during processing
- Leads to slower response times
- Creates scalability challenges

4.1.3 Example: Multiple Clients

WSGI Limitation

Scenario: 5 clients send requests simultaneously

What happens:

1. Web server receives all 5 HTTP requests

2. WSGI starts processing Request 1
3. Requests 2, 3, 4, 5 wait in queue
4. Only after Request 1 completes, Request 2 starts
5. This continues sequentially for all requests

Result: High latency and poor performance under load

4.1.4 Werkzeug

Werkzeug Library

Werkzeug is a comprehensive WSGI web application library.

Role: Implements the WSGI protocol for Flask

Note: When you install Flask, Werkzeug is automatically installed as a dependency.

4.1.5 Gunicorn Web Server

Gunicorn (Green Unicorn)

Gunicorn is a WSGI HTTP server specifically designed for Python web applications.

Known for:

- Efficiency in standard use cases
- Decent scalability
- Wide adoption in Flask applications

Warning

Gunicorn Limitations:

Being WSGI-based, Gunicorn suffers from:

- Synchronous processing bottlenecks
- I/O wait times
- High latency under concurrent load
- Performance issues at scale

4.1.6 Flask API Code

Flask API code is written in standard, synchronous Python:

- Processes one request at a time
- No built-in concurrency support
- Blocking operations halt execution

4.2 Flask Architecture Summary

Component	Flask Choice
SGI Protocol	WSGI (Synchronous)
WSGI Implementation	Werkzeug
Web Server	Gunicorn
API Code	Synchronous Python
Processing Model	Sequential/Blocking

Important Note

Key Takeaway: Flask's entire pipeline is synchronous, which means:

- Only one request processed at a time
- Other requests wait in queue
- Performance degrades with concurrent users

4.3 FastAPI Architecture

FastAPI takes a fundamentally different approach by using asynchronous components throughout the stack.

4.3.1 FastAPI Components

1. **SGI Protocol**: ASGI (Asynchronous Server Gateway Interface)
2. **ASGI Implementation**: Starlette library
3. **Web Server**: Uvicorn
4. **API Code**: Async/Await Python code

4.3.2 ASGI (Asynchronous Server Gateway Interface)

What is ASGI?

ASGI is a modern, asynchronous interface for Python web applications.

Key Characteristics:

- Newer protocol designed for modern needs
- **Asynchronous** in nature
- Processes multiple requests concurrently
- **Non-blocking architecture**

ASGI vs WSGI

Google Search Result:

"WSGI is an older synchronous interface for Python web applications, while ASGI is a newer asynchronous interface that's better suited for modern web applications like those using WebSockets and real-time features."

The Biggest Advantage of ASGI:

- Can handle **concurrent requests**
- Multiple requests processed in parallel
- Non-blocking I/O operations
- Superior performance under load

4.3.3 Starlette

Starlette Library

"The little ASGI framework that shines."

Role: Implements ASGI protocol for FastAPI

Capabilities:

- Asynchronous request handling

- Concurrent processing support
- High-performance HTTP operations

4.3.4 Uvicorn Web Server

Uvicorn

Uvicorn is a high-performance ASGI server.

Comparison with Gunicorn:

- Uvicorn: ASGI server (asynchronous)
- Gunicorn: WSGI server (synchronous)

Performance Comparison

Search Result: "Uvicorn vs Gunicorn"

"Uvicorn and Gunicorn are both Python web servers, but they cater to different use cases:

- **Uvicorn**: High-performance ASGI server (NOT WSGI)
- **Gunicorn**: Mature WSGI server

Uvicorn is generally preferred for its high performance and asynchronous capabilities."

4.3.5 Async/Await in Python

FastAPI supports Python's `async/await` syntax, enabling true asynchronous processing.

How Async/Await Works

Scenario: API endpoint calls ML model for prediction

Without async/await (Synchronous):

1. Request arrives at API
2. API sends input to ML model
3. **API waits** while model processes (blocking)
4. During wait time, **no other requests can be handled**
5. Model returns prediction
6. API sends response
7. **Only now** can next request be processed

With async/await (Asynchronous):

1. Request arrives at API
2. API sends input to ML model with `await`

3. **API continues accepting other requests** while model processes
4. When model finishes, API sends response
5. **Multiple requests processed concurrently**

```python
# Without async/await (Blocking)
def predict(features):
    result = ml_model.predict(features)  # Blocks here
    return result

# With async/await (Non-blocking)
async def predict(features):
    result = await ml_model.predict(features)  # Doesn't block
    return result
```

Listing 1: Async/Await Example

4.4 FastAPI Architecture Summary

Component	FastAPI Choice
SGI Protocol	ASGI (Asynchronous)
ASGI Implementation	Starlette
Web Server	Uvicorn
API Code	Async/Await Python
Processing Model	Concurrent/Non-blocking

4.5 Side-by-Side Comparison

Aspect	Flask	FastAPI
SGI Protocol	WSGI (Sync)	ASGI (Async)
Implementation	Werkzeug	Starlette
Web Server	Gunicorn	Uvicorn
Processing	Sequential	Concurrent
Blocking	Yes	No
Performance	Moderate	High
Scalability	Limited	Excellent
Latency	Higher	Lower

4.6 The Restaurant Analogy

To understand the difference between Flask and FastAPI in simple terms, let's use a restaurant analogy:

Flask as a Waiter

Flask is like a waiter who:

1. Takes order from Customer 1
2. Goes to kitchen and gives order to chef
3. **Stands in kitchen waiting** while food is being prepared
4. Takes prepared food back to Customer 1
5. **Only then** goes to Customer 2
6. Repeats the same process

Problem: Time is wasted waiting in the kitchen. Other customers must wait unnecessarily.

FastAPI as a Waiter

FastAPI is like a waiter who:

1. Takes order from Customer 1
2. Goes to kitchen and gives order to chef
3. **Knows food will take time, so returns to take more orders**
4. Takes order from Customer 2
5. Gives it to kitchen
6. Meanwhile, Customer 1's order is ready
7. Delivers Customer 1's food
8. Continues this asynchronous pattern

Benefit: Multiple customers served efficiently. No waiting time wasted.

Important Note

Key Difference:

- **Flask (Synchronous)**: Waits idle during processing, blocks other requests
- **FastAPI (Asynchronous)**: Serves multiple customers concurrently, maximizes efficiency

This is why FastAPI is **Fast to Run**!

5 Why FastAPI is Fast to Code

5.1 Three Key Aspects

FastAPI makes development faster through three main features:

1. Automatic Input Validation
2. Auto-Generated Interactive Documentation
3. Seamless Modern Library Integration

5.2 1. Automatic Input Validation

5.2.1 The Problem in Python

Python has dynamic typing by default:

- Variables are created dynamically
- Types can change at runtime
- No built-in type checking
- A variable can be integer, then string, etc.

Dynamic Typing Example

```
# Python allows this:
x = 5            # x is integer
x = "hello"      # now x is string
x = [1, 2, 3]    # now x is list

# No errors - but can cause issues in production!
```

5.2.2 Why This Matters for APIs

When building enterprise-level APIs, you need to ensure:

- Input data is in correct format
- Types match expectations
- Values are within valid ranges
- No type-related runtime errors

5.2.3 FastAPI + Pydantic Solution

FastAPI integrates tightly with Pydantic for automatic validation:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# Define expected input structure
class PredictionInput(BaseModel):
```

```python
    feature1: float  # Must be float
    feature2: int    # Must be integer
    feature3: str    # Must be string

@app.post("/predict")
def predict(input_data: PredictionInput):
    # If data doesn't match model, automatic error!
    # No manual checking needed
    return {"prediction": "result"}
```

Listing 2: Automatic Validation Example

Important Note

Behind the Scenes:

When a request arrives:

1. Pydantic automatically checks all types
2. Validates data format
3. Converts types if possible
4. Returns clear error if validation fails

All without writing manual validation code!

5.3 2. Auto-Generated Interactive Documentation

5.3.1 The Documentation Challenge

In traditional API development:

- Building the software is one part
- Creating documentation is equally important
- Users need docs to understand how to use your API
- Documentation takes significant time and effort

For APIs, documentation must include:

- Purpose of each endpoint
- Input format expected
- Output format returned
- Example requests and responses
- Error codes and meanings

5.3.2 FastAPI's Solution

Automatic Documentation

As you write API code, FastAPI automatically generates:

- Complete API documentation
- **Interactive interface** to test endpoints
- All endpoint details
- Request/response schemas

Accessing Documentation:

Simply navigate to: `http://your-api-url/docs`

Interactive Documentation Features

The auto-generated docs allow you to:

1. View all API endpoints
2. See expected input/output formats
3. **Test endpoints directly in browser**
4. See real-time responses
5. Understand data models

No need for Postman or other testing tools!

5.4 3. Seamless Modern Library Integration

FastAPI is designed with modern development in mind and integrates seamlessly with:

5.4.1 Machine Learning Libraries

- **scikit-learn**: Traditional ML models
- **TensorFlow**: Deep learning frameworks
- **PyTorch**: Neural networks
- **XGBoost, LightGBM**: Gradient boosting

5.4.2 Authentication

- **OAuth**: Social login integration
- **JWT**: Token-based authentication
- Built-in security utilities

5.4.3 Database Integration

- **SQLAlchemy**: SQL databases
- **MongoDB**: NoSQL databases
- **Redis**: Caching solutions

5.4.4 Deployment

- **Docker**: Containerization
- **Kubernetes**: Orchestration
- **AWS, GCP, Azure**: Cloud platforms

Important Note

Why This Matters:

Modern applications require integration with multiple services. FastAPI's tight coupling with these libraries means:

- Less boilerplate code
- Fewer compatibility issues
- Faster development
- Better maintainability

5.5 Summary: Fast to Code

Feature	Benefit
Auto Validation	No manual type checking code
Auto Docs	No separate documentation effort
Modern Integration	Seamless library compatibility
Type Hints	Better IDE support and fewer bugs
Async Support	Write concurrent code easily

6 Installing and Setting Up FastAPI

6.1 Prerequisites

Before installing FastAPI, ensure you have:

- **Python 3.8+**: FastAPI requires Python 3.8 or higher
- **pip**: Python package manager
- **VS Code** (recommended): Or any code editor
- **Terminal/Command Prompt**: For running commands

6.2 Step-by-Step Setup

6.2.1 Step 1: Create Project Folder

```
# Navigate to desired location (e.g., Desktop)
# Create new folder
mkdir fastapi_tutorials
cd fastapi_tutorials
```

6.2.2 Step 2: Open in VS Code

```
# Open VS Code
code .
```

Or manually open VS Code and open the folder.

6.2.3 Step 3: Create Virtual Environment

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

Important Note

Why Virtual Environment?

Virtual environments:

- Isolate project dependencies
- Prevent version conflicts
- Make projects portable
- Are Python best practice

6.2.4 Step 4: Activate Virtual Environment

On Windows:

```
myenv\Scripts\activate.bat
```

On Linux/Mac:

```
source myenv/bin/activate
```

You should see `(myenv)` in your terminal prompt.

6.2.5 Step 5: Install Required Packages

```
# Install FastAPI, Uvicorn, and Pydantic
pip install fastapi uvicorn pydantic
```

What gets installed:

- `fastapi`: The FastAPI framework
- `uvicorn`: ASGI web server
- `pydantic`: Data validation library
- `starlette`: Auto-installed with FastAPI

Wait for installation to complete. You'll see Starlette being installed automatically as it's a FastAPI dependency.

6.3 Verifying Installation

```
# Check installed packages
pip list

# Look for:
# fastapi
# uvicorn
# pydantic
# starlette
```

7 Building Your First FastAPI Application

7.1 Hello World API

Let's build a simple "Hello World" API to understand the basics.

7.1.1 Step 1: Create Python File

Create a new file named `main.py` in your project folder.

7.1.2 Step 2: Write the Code

```python
from fastapi import FastAPI

# Create FastAPI app instance
app = FastAPI()

# Define route and endpoint
@app.get("/")
def hello():
    return {"message": "Hello World"}
```

Listing 3: main.py - Hello World API

7.2 Code Explanation

Let's break down each part:

7.2.1 1. Import FastAPI

```python
from fastapi import FastAPI
```

- Imports the `FastAPI` class
- This is the main class for creating API applications

7.2.2 2. Create App Instance

```python
app = FastAPI()
```

- Creates an instance of FastAPI class
- This `app` object is your entire API application
- All routes and endpoints are registered with this object

7.2.3 3. Define Route

```python
@app.get("/")
```

This is a **decorator** that:

- Defines an API route/path
- `get` specifies HTTP GET method
- `"/"` is the URL path (home route)

HTTP Methods

GET: Retrieve/fetch data from server

POST: Send/create data on server

We use GET here because we're fetching data from the API.

7.2.4 4. Define Handler Function

```
def hello():
    return {"message": "Hello World"}
```

- Function executes when route is accessed
- Returns a Python dictionary
- FastAPI automatically converts to JSON

7.3 Running the API

7.3.1 Start the Server

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

Command breakdown:

- `uvicorn`: The ASGI server
- `main`: Python filename (main.py)
- `app`: FastAPI object name
- `--reload`: Auto-reload on code changes

Important Note

Why –reload?

The `--reload` flag:

- Watches for code changes
- Automatically restarts server
- Great for development
- Don't use in production!

7.3.2 Expected Output

```
INFO:     Uvicorn running on http://127.0.0.1:8000
INFO:     Started server process
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

7.3.3 Test the API

Open your browser and navigate to:

`http://127.0.0.1:8000/`

You should see:

```
{"message": "Hello World"}
```

7.4 Adding More Endpoints

Let's add another endpoint to understand multiple routes.

7.4.1 Updated Code

```python
from fastapi import FastAPI

app = FastAPI()

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

@app.get("/about")
def about():
    return {"message": "CampusX is an education platform where you can learn AI"}
```

Listing 4: main.py - Multiple Endpoints

7.4.2 What Changed?

We added a new endpoint:

- **Route**: `/about`
- **Method**: GET
- **Function**: `about()`
- **Response**: Information about CampusX

7.4.3 Auto-Reload Magic

Important Note

Because we used `--reload`, the server automatically:

1. Detected the code change
2. Restarted the server
3. Loaded the new endpoint

No need to manually restart!

7.4.4 Testing Multiple Endpoints

Test Home Endpoint:

`http://127.0.0.1:8000/`

Response:

```
{"message": "Hello World"}
```

Test About Endpoint:

`http://127.0.0.1:8000/about`

Response:

```
{"message": "CampusX is an education platform where you can learn AI"}
```

7.5 Understanding Routes

How Routes Work

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

Routes:

- `/` → Home endpoint
- `/about` → About endpoint
- Can add any route: `/users`, `/products`, etc.

Each route connects to a specific function that handles the request.

8 Interactive API Documentation

8.1 Accessing Auto-Generated Docs

FastAPI automatically creates interactive documentation for your API.

8.1.1 Swagger UI Documentation

Navigate to:

`http://127.0.0.1:8000/docs`

8.2 What You'll See

The documentation page displays:

1. **All Endpoints**: List of all API routes
2. **HTTP Methods**: GET, POST, etc. for each endpoint
3. **Parameters**: Expected inputs for each endpoint
4. **Response Models**: Output structure
5. **Try It Out**: Interactive testing interface

8.3 Example Documentation View

For our current API, you'll see:

```
GET /
    Returns: {"message": "Hello World"}

GET /about
    Returns: {"message": "CampusX is an education..."}
```

8.4 Interactive Testing

8.4.1 Step 1: Expand Endpoint

Click on any endpoint (e.g., `GET /`) to expand it.

8.4.2 Step 2: Click "Try it out"

You'll see a button labeled **"Try it out"**. Click it.

8.4.3 Step 3: Execute

Click the **"Execute"** button.

8.4.4 Step 4: View Response

The documentation will show:

```
Response Body:
{
  "message": "Hello World"
```

```
}

Response Headers:
content-type: application/json
content-length: 26
date: ...
server: uvicorn

Status Code: 200 OK
```

8.5 Benefits of Auto-Documentation

1. **No Extra Work**: Documentation generated automatically
2. **Always Up-to-Date**: Updates with code changes
3. **Interactive**: Test endpoints directly
4. **Clear Structure**: Well-organized and readable
5. **No Postman Needed**: Built-in testing
6. **Team Collaboration**: Easy for others to understand your API

Important Note

Amazing Feature:
As you add more endpoints, parameters, and data models, the documentation automatically updates. You never have to manually write API docs!

8.6 Alternative Documentation

FastAPI also provides ReDoc documentation:

Navigate to:

`http://127.0.0.1:8000/redoc`

This provides an alternative, cleaner documentation view.

9 FastAPI Core Concepts

9.1 Understanding Decorators

9.1.1 What is a Decorator?

In Python, a decorator is a function that modifies another function's behavior.

```
@app.get("/")   # This is a decorator
def hello():    # This is the function being decorated
    return {"message": "Hello"}
```

Listing 5: Decorator Syntax

9.1.2 What @app.get() Does

- Registers the function as an API endpoint
- Associates it with HTTP GET method
- Maps it to the specified URL path
- Tells FastAPI to execute this function when route is accessed

9.2 HTTP Methods in FastAPI

9.2.1 Common HTTP Methods

Method	Decorator	Purpose
GET	`@app.get()`	Retrieve/fetch data
POST	`@app.post()`	Create/send new data
PUT	`@app.put()`	Update existing data (full)
PATCH	`@app.patch()`	Update existing data (partial)
DELETE	`@app.delete()`	Delete data

9.2.2 GET vs POST

When to Use Each

Use GET when:

- Fetching data from server
- Reading information
- No data modification
- Example: Get user profile, search results

Use POST when:

- Sending data to server
- Creating new resources
- Submitting forms
- Example: User registration, file upload

9.3 Return Values

9.3.1 Automatic JSON Conversion

FastAPI automatically converts Python dictionaries to JSON:

```
@app.get("/")
def hello():
    # You return a Python dict
    return {"message": "Hello", "status": "success"}

# FastAPI automatically converts to JSON:
# {"message": "Hello", "status": "success"}
```

9.3.2 What You Can Return

- **Dictionary**: Converted to JSON object
- **List**: Converted to JSON array
- **Pydantic Models**: Converted to JSON with validation
- **Primitive Types**: strings, numbers, booleans

9.4 Path Parameters

You can make routes dynamic using path parameters:

```
@app.get("/user/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id, "name": "John Doe"}

# Access: http://127.0.0.1:8000/user/123
# Response: {"user_id": 123, "name": "John Doe"}
```

Listing 6: Path Parameters Example

9.5 FastAPI Application Object

9.5.1 The app Instance

```
app = FastAPI()
```

This `app` object:

- Is your entire API application
- Registers all routes and endpoints
- Handles request routing
- Manages middleware and dependencies
- Can be customized with metadata

9.5.2 Customizing the App

```
app = FastAPI(
    title="My API",
    description="This is my awesome API",
    version="1.0.0"
)
```

Listing 7: Custom App Configuration

This information appears in the auto-generated documentation!

10 Project Organization and Best Practices

10.1 Recommended Project Structure

```
fastapi_tutorials/
|-- myenv/                  # Virtual environment
|-- main.py                 # Main application file
|-- requirements.txt        # Dependencies
|-- .gitignore             # Git ignore file
|-- README.md              # Project documentation
+-- app/                    # Application package (future)
    |-- __init__.py
    |-- models.py           # Data models
    |-- routes.py           # API routes
    +-- config.py           # Configuration
```

10.2 Creating requirements.txt

To make your project reproducible:

```
# Generate requirements file
pip freeze > requirements.txt
```

This allows others to install dependencies:

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

10.3 Development Best Practices

1. **Use Virtual Environments**: Always isolate project dependencies
2. **Use –reload in Development**: Auto-restart on code changes
3. **Organize Code**: Separate routes, models, and logic as project grows
4. **Write Descriptive Names**: Clear function and route names
5. **Use Type Hints**: Helps with auto-completion and validation
6. **Test Endpoints**: Use the interactive docs for testing
7. **Version Control**: Use Git to track changes

10.4 Common Commands Reference

Essential Commands

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

# Activate virtual environment
myenv\Scripts\activate.bat          # Windows
source myenv/bin/activate           # Linux/Mac

# Install packages
pip install fastapi uvicorn pydantic

# Run FastAPI server
uvicorn main:app --reload

# Access API
http://127.0.0.1:8000/

# Access documentation
http://127.0.0.1:8000/docs

# Stop server
Ctrl + C
```