Pre requisite

16 November 2025 22:22

1. **Basics of Python**
2. **Basics of HTTP and its working**
3. **HTTP Request and Response cycle**
4. **Install Docker**

What is an API?

15 November 2025 20:46

What is an API?

APIs are mechanisms that enable two software components—such as the frontend and backend of an application—to communicate with each other using a defined set of rules, protocols, and data formats.

- An API (Application Programming Interface) is essentially a set of rules and protocols that allows different software applications to communicate with each other
- It can be thought of as a bridge that connects different systems or services, enabling them to share data and functionalities without directly interacting with each other's underlying code
- Ex: Weather app

2. Key Features

Sunday, January 19, 2025 8:01 AM

- Enable different systems, platforms, and applications to communicate and share data effortlessly
- Significantly reduce the time and effort required to develop applications by leveraging pre-built functionalities
- Enable systems to scale and adapt to growing or changing requirements
- APIs contribute to creating dynamic and responsive applications that offer better user experiences
- Allow organizations to expand their ecosystem and collaborate with external developers and partners
- APIs provide controlled and secure access to data, allowing organizations to share information with clients, partners, or the public
- APIs provide mechanisms to enforce security policies and ensure compliance with industry standards
- APIs can help organizations reduce operational and development costs

Types of APIs

16 November 2025 19:43

1. **Web API:**
   - APIs that are accessible over the web using HTTP/HTTPS protocols
   - Allow applications to communicate over the internet
   - Data is usually shared in JSON/XML formats
   - Examples:
     - *Fetching weather data from OpenWeather API*
     - *Embedding Google Maps on a website*

2. **Library API:**
   - APIs provided by libraries or frameworks to expose specific functionality for developers
   - Allow developers to utilize predefined functions or methods without implementing them from scratch
   - Examples:
     - *NumPy API: Provides functions for numerical computations*
     - *TensorFlow API: Offers tools for building and training ML/DL models*
     - *Matplotlib API: Allows creating visualizations programmatically*

3. **Remote API:**
   - APIs designed to interact with systems located on a different network or server, often through the internet or intranet
   - Web API is a subset of Remote API, i.e., all Web APIs are Remote APIs, but not all Remote APIs are Web APIs
   - Makes use of Intranet APIs, which can only be accessible within a private network
   - Examples:
     - *Deploying virtual machines using AWS EC2 API*
     - *Retrieving remote files stored in Google Drive through its API*

4. **Database API:**
   - APIs that allow applications to interact with databases
   - Provide a structured way to perform CRUD (Create, Read, Update, Delete) operations on a database
   - Examples:
     - *MySQL Connector API: Enables interaction with MySQL databases*
     - *MongoDB API: Allows CRUD operations on NoSQL data*
     - *Firebase Realtime Database API: Facilitates real-time database interactions*

5. **Hardware API:**
   - APIs that enable software to interact with physical hardware devices
   - Provide an abstraction layer to control and retrieve data from devices without needing low-level programming
   - Examples:
     - *GPU APIs like CUDA or OpenCL for performing parallel computations*
     - *APIs for IoT devices such as sensors or smart appliances*
     - *Using APIs to control drones or robots programmatically*

6. **GUI API:**
   - APIs designed for building and interacting with graphical user interfaces
   - Allow developers to create and manage GUI components programmatically
   - Examples:
     - *Java Swing API: Provides tools to create desktop GUIs in Java*
     - *Tkinter: A Python library for creating GUI applications*
     - *Android SDK: Enables developers to build mobile app interfaces*

API Protocols

16 November 2025 20:09

1. REST:

- Abbreviation for Representational State Transfer
- It's a lightweight architectural style that uses HTTP for communication
- Follow a set of principles, such as statelessness and resource representation using URLs
- Each API call is independent of the other — no memory of prev call.
- Common HTTP methods: GET, POST, PUT, DELETE
- Use Cases: Web applications, mobile applications, and public APIs (e.g., Twitter API, GitHub API)

```
GET /users/123 HTTP/1.1
Host: example.com
```

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

2. SOAP:

- Abbreviation for Simple Object Access Protocol
- A protocol that relies on XML messaging for communication
- Designed for more structured and secure interactions
- Has built-in error handling and security (e.g., WS-Security)

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetUserDetails>
      <UserId>123</UserId>
    </GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

3. GraphQL:

- Allows clients to request only the data they need, reducing over-fetching and under-fetching
- Clients specify the structure of the response
- All queries are handled at one endpoint

```
{
  user(id: "123") {
    name
    email
    posts {
      title
      comments {
        text
      }
    }
  }
```

4. gRPC:

- Abbreviation for Google Remote Procedure Call
- A high-performance RPC framework by Google using Protocol Buffers (Protobuf) for message serialization
- Operates over HTTP/2, providing bi-directional streaming (bidirectional)
- Works across various programming languages and minimizes payload size

```
service UserService {
  rpc GetUserDetails (UserRequest) returns (UserResponse);
}
message UserRequest {
  int32 user_id = 1;
}
message UserResponse {
  int32 user_id = 1;
  string name = 2;
  string email = 3;
}
```

5. WebSocket:

- A protocol providing full-duplex communication over a single TCP connection
- Enables continuous exchange of data without re-establishing the connection
- Ideal for real-time use cases due to its low latency
- Use Cases: Chat applications, online gaming, live data feeds

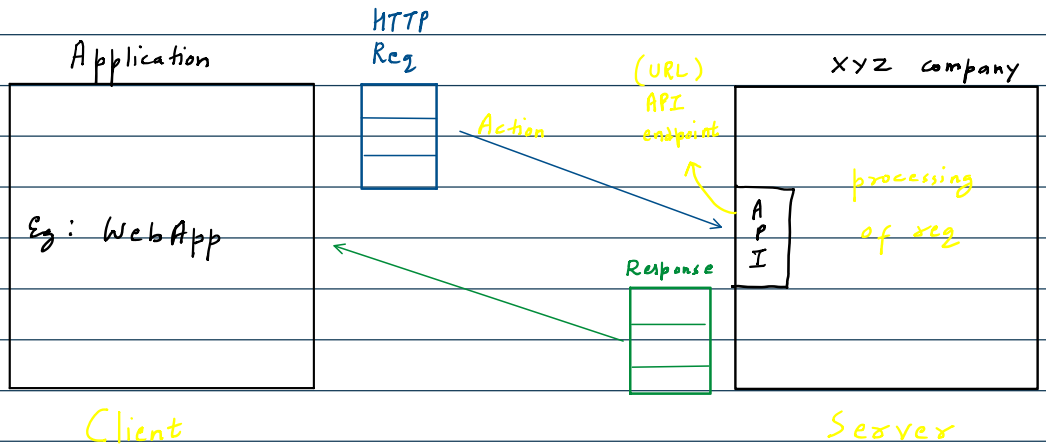
```
const socket = new WebSocket("ws://example.com/stocks");
socket.onmessage = (event) => {
  console.log("Stock update:", event.data);
};
```

Feature	REST	SOAP	GraphQL
Data Format	JSON, XML, etc.	XML only	JSON only
Flexibility	High	Low	Very High
Data Format	JSON, XML, etc.	XML only	JSON only
Flexibility	High	Low	Very High

Data Format	JSON, XML, etc.	XML only	JSON only
Flexibility	High	Low	Very High
Performance	Fast	Slower	Efficient
Use Case	Modern web APIs	Enterprise applications	Dynamic client needs

Working

16 November 2025 20:11



1. **Request Initiation:**
   - The process begins when a client (like a web browser, mobile app, etc.) initiates a request to the API
   - This request is usually made using HTTP methods such as GET, POST, PUT, or DELETE
     action
2. **API Endpoint:**
   - The request is sent to a specific URL known as an API endpoint (URL)
   - This endpoint is like a door that the API has opened for requests
3. **Request Processing:**
   - The server receives the request and processes it
   - This involves verifying the request parameters, checking authentication, and accessing databases, etc.
4. **Response Generation:**
   - After processing the request, the server generates a response
   - This response typically includes the requested data or a confirmation of the action performed
5. **Response Delivery:**
   - The response is sent back to the client
   - The client receives the response and uses the data to perform the desired action or display information to the user.

API Components

16 November 2025 20:11

1. Endpoint:

- It's a specific URL where the API can be accessed by a client to perform a certain action
- Acts as the communication touchpoint between the client and the server
- Usually include path parameters and query parameters

location

https://www.iplt20.com/news

for filtering data

Eg: XYZ company

WWW.XYZ.com/employee?dept=sales&location=delhi

Domain Name | path parameter | query parameter

2. Request:

- Message sent by a client to the API to ask for information or perform an operation
- Key components of a request:
  - Method: Specifies the type of operation (e.g., GET, POST, PUT, DELETE)
  - Headers: Provide metadata about the request, such as content type
  - Body: Contains the data being sent to the server

R C U D

method | endpoint | protocol version

```
POST /users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer token123
Body:
{
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

headers

body

3. Response: (generated at server side)

- message sent back by the API after processing a client's request
- Key components of a response:
  - Status Code: Indicates the outcome of the request (e.g., success, error)
  - Headers: Metadata about the response, such as content type or caching instructions
  - Body: The actual data being returned, typically in JSON, XML

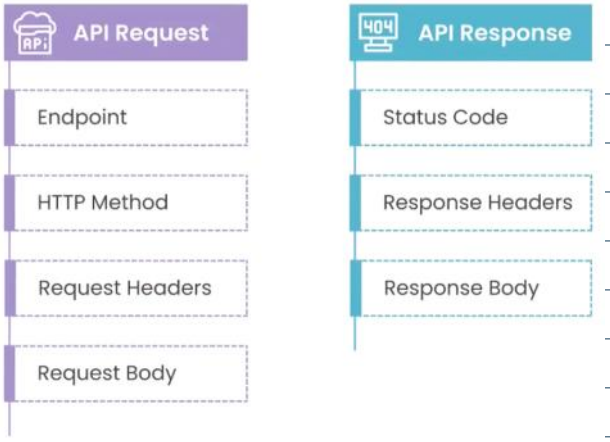
protocol version | status code

```
HTTP/1.1 200 OK
Content-Type: application/json
Body:
{
  "id": 123,
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

] Headers

Body

200 → success



4. **Rate Limiting and Quotas:**
   - Mechanisms to control the number of requests a client can make within a specified time
   - Mainly to prevent abuse and ensure fair usage
   - Headers for implementation:
     - X-RateLimit-Limit: Maximum requests allowed
     - X-RateLimit-Remaining: Requests remaining in the current window
     - Retry-After: Time to wait before retrying

API Lifecycle

16 November 2025 20:11

- The API lifecycle refers to the comprehensive process of designing, developing, deploying, managing, and eventually retiring an API
- It is a structured framework that ensures APIs are effective, scalable, secure, and meet business objectives
- Understanding the API lifecycle is critical for maintaining high-quality API products and enhancing the user experience

1. **Planning and Design:**
   - Identify business goals, user needs, and the problems the API will solve
   - Determine the target audience: developers, partners, or internal teams
   - Define endpoints, request/response formats, and data models
   - Follow design methodologies like REST, GraphQL, or gRPC
   - Ensure adherence to API design best practices, such as intuitive endpoints, consistent naming, and proper versioning
   - **Tools:** Postman, SwaggerHub, and Stoplight for API design and documentation

2. **Development:**
   - The development phase involves implementing the API's backend logic and infrastructure
   - Write the server-side code using programming frameworks like Flask, FastAPI, or Express.js
   - Integrate authentication and security mechanisms
   - Perform unit testing to validate individual functions and methods
   - **Tools:** Git, Jenkins, Docker, or Kubernetes for CI/CD pipelines

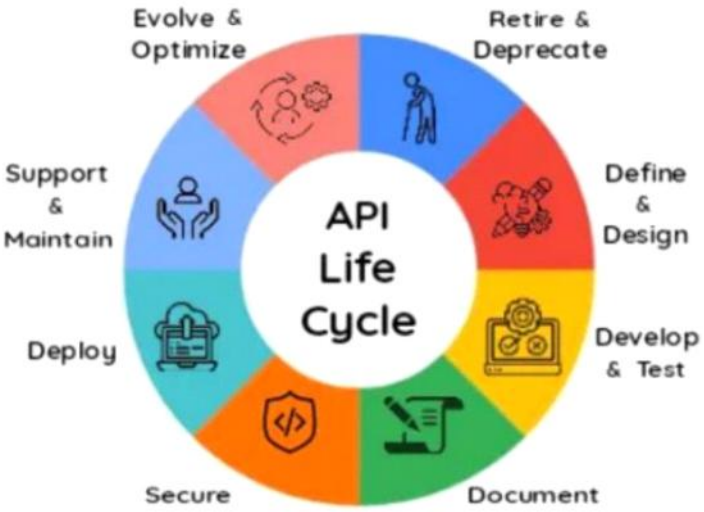
3. **Deployment:**
   - Involves releasing the API into a production environment where users can access it
   - Deploy the API to staging, testing, and production environments
   - Use cloud services like AWS, Azure, or GCP for scalability and reliability
   - Set up an API gateway to manage routing, caching, and load balancing
   - Popular gateways include AWS API Gateway, Kong, and Apigee

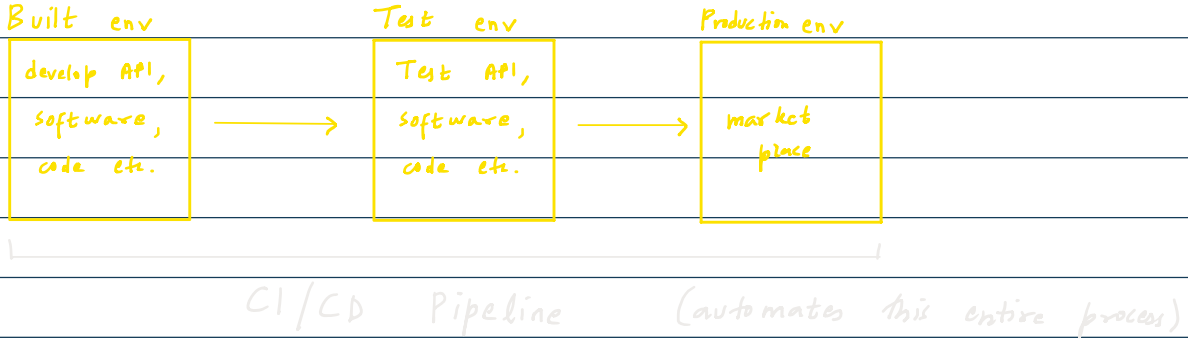
4. **Monitoring and Management:**
   - Track performance metrics like response times, uptime, and error rates
   - Gather usage data to understand traffic patterns, popular endpoints, and user behavior
   - Identify areas for improvement or optimization
   - Enforce rate limiting to prevent abuse
   - **Tools:** Datadog, New Relic, and Grafana

5. **Updates and Versioning:**
   - Ensure new updates do not break existing clients
   - Provide clear migration paths if breaking changes are unavoidable

5. **Updates and Versioning:**
   - Ensure new updates do not break existing clients
   - Provide clear migration paths if breaking changes are unavoidable
   - Notify users in advance of deprecated features or versions
   - Provide timelines and guidelines for transition

6. **Retirement:**
   - When an API no longer serves its purpose, it is retired or deprecated
   - Inform stakeholders and users about the API's retirement well in advance
   - Provide tools or resources to help users transition to newer APIs
   - Archive or securely dispose of any stored data associated with the API





Authentication & Authorization

16 November 2025 20:11

- When building an API, especially one that will be exposed to the public or clients, managing who can access it and ensuring the security of data is paramount
- Authentication and authorization are two critical concepts that help in securing APIs and ensuring that only legitimate users can access resources

Authentication:

- Process of verifying the identity of a user or system
- It answers the question: ***Who are you?***

Authorization:

- process of determining what an authenticated user is allowed to do
- It answers the question: ***What can you do?***

Types of Authentication Mechanisms:

1. **API Keys:**
   - They work by sending a key (usually a long string of characters) with each request to identify the client
   - Typically used for public APIs or services where the user is not required to log in manually
   - Easy to implement and widely supported
   - Can be intercepted if not transmitted over HTTPS and does not provide strong user verification
2. **OAuth (Open Authorization):**
   - Open standard for access delegation commonly used for third-party authentication
   - Allows a user to grant a third-party application access to their resources without sharing their credentials
   - Secure and allows fine-grained permissions
   - Comparatively more complex to implement
3. **JWT (JSON Web Tokens):**
   - JWT is a compact and self-contained method for securely transmitting information between parties as a JSON object
   - It's commonly used for handling user authentication in stateless applications
   - No need to store session information server-side
   - Tokens can become stale or vulnerable if not properly managed
4. **Bearer Tokens:**
   - They are a form of access token commonly used in API requests to authorize access to protected resources
   - These tokens are typically provided by an OAuth server or after a successful login
   - Secure and stateless authentication method

Types of Authorization Mechanisms:

1. **Role-Based Access Control (RBAC):**
   - Common authorization method used to assign permissions based on user roles
   - The system defines roles (e.g., Admin, User, Manager) and each role has certain permissions associated with it
   - After a user is authenticated, the API checks the user's role and grants or denies access based on predefined role permissions
   - Simple to manage when there are clear roles

2. **OAuth 2.0 and OpenID Connect (OIDC):**
   - OAuth 2.0 is a widely used framework for authorization
   - OIDC is an identity layer on top of OAuth 2.0, which provides authentication features, in addition to OAuth's authorization features
   - OAuth 2.0 is typically used for delegating access to APIs on behalf of a user

Best Practices for API Authentication and Authorization:

- Ensure all authentication and authorization requests are transmitted over HTTPS to prevent **man-in-the-middle (MITM)** attacks
- Use hashing algorithms (e.g., bcrypt, Argon2) to store user passwords
- Set appropriate expiration times for tokens and use refresh tokens to allow users to renew their sessions without needing to re-authenticate
- Implement **multi-factor authentication (MFA)** where possible to increase the security of APIs
- Return generic error messages for failed authentication or authorization attempts to avoid exposing sensitive information
- Regularly review API logs to detect suspicious activity and to identify potential security breaches

What is FastAPI?

16 November 2025 19:20

- FastAPI is a modern, high-performance web framework for building APIs with Python based on standard Python type hints
- It is designed to make it easy and fast to build efficient, production-ready APIs, especially those involving asynchronous I/O operations and data validation
- Requires Python 3.7+

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.

The key features are:

- **Fast**: Very high performance, on par with **NodeJS** and **Go** (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- **Fast to code**: Increase the speed to develop features by about 200% to 300%. *
- **Fewer bugs**: Reduce about 40% of human (developer) induced errors. *
- **Intuitive**: Great editor support. Completion everywhere. Less time debugging.
- **Easy**: Designed to be easy to use and learn. Less time reading docs.
- **Short**: Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- **Robust**: Get production-ready code. With automatic interactive documentation.
- **Standards-based**: Based on (and fully compatible with) the open standards for APIs: OpenAPI [↪] (previously known as Swagger) and JSON Schema [↪].

1. **High Performance:**
   - One of the fastest Python web frameworks
   - Comparable in speed to Node.js and Go for many API tasks
2. **Based on Python Type Hints:**
   - Validate inputs automatically
3. **Built-in Data Validation:**
   - Ensures that all input data is structured, typed, and clean
4. **Asynchronous Support:**
   - Designed from the ground up to support async I/O operations
5. **Easy Testing & Debugging:**
   - Works great with pytest and testing tools
   - Return types are predictable and readable for debugging

type hints

return type

practice.py practice.py\...

```
1  def add_integers(a: int, b: int) -> int:
2      return a + b
3
4
5  print(add_integers(3, 4))
```

practice.py practice.py\...

```
1  def add_integers(a: int, b: int) -> int:
2      return a + b
3
4
5  print(add_integers(3.5, 5.5))
```

ans = 9.0 & not error

* python does not enforce type hints.

6. **Automatic API Documentation:**
   - Built-in integration with **Swagger UI (/docs)** and **ReDoc (/redoc)**
   - Helps both backend and frontend teams explore and test endpoints easily

7. **Ideal for ML & Data Science Projects:**
   - Wrapping ML models as APIs (**/predict**)
   - Serving pre/post-processing logic
   - Managing model versions and feature validation

Architecture

16 November 2025 23:42

(ASGI Architecture)

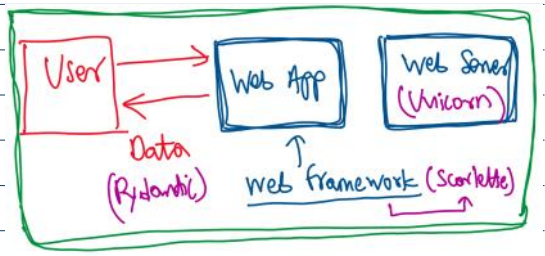
FastAPI is Built on Three Core Libraries:

- **Starlette:** ✓ Asynchronous Server Gateway Interface
  - Lightweight ASGI web framework/toolkit used as the **core web layer** in FastAPI
  - Handles the registration and dispatching of HTTP routes (e.g., GET, POST)
  - Enables real-time WebSocket communication

- **Pydantic:** ✓
  - Data parsing and validation library that leverages Python's type hints to enforce and transform data structures
  - Ensures request bodies, query params, headers, and path params match expected types
  - Converts data (e.g., string to int, string to datetime) before it reaches your function
  - Helps auto-generate JSON Schema for documentation

- **Uvicorn:**
  - A lightning-fast ASGI server implementation based on uvloop and httptools
  - Launches FastAPI apps and handles incoming client requests
  - Communicates with FastAPI via the ASGI specification
  - Supports async I/O, which makes it fast and scalable

- Starlette – Web framework
- Pydantic – Data processing/validation
- Uvicorn – Web Server



Installation

16 November 2025 23:42

1. **Create/Activate a virtual environment:**
2. **Install FastAPI and Uvicorn:**
   - pip install fastapi uvicorn

```
(base) D:\Coding\fastAPI\demo>conda create -n fastAPIdemo python=3.11
3 channel Terms of Service accepted
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

```
(base) D:\Coding\fastAPI\demo>conda activate fastAPIdemo

(fastAPIdemo) D:\Coding\fastAPI\demo>python -m pip install fastapi uvicorn
Collecting fastapi
```

```
(fastAPIdemo) D:\Coding\fastAPI\demo>uvicorn --version
Running uvicorn 0.38.0 with CPython 3.11.14 on Windows

(fastAPIdemo) D:\Coding\fastAPI\demo>code .
```

→ to open VS code

First App using FastAPI

16 November 2025 23:42

```python
from fastapi import FastAPI
app = FastAPI()
@app.get('/')
def index():
    return {'message': 'Hello, FastAPI!'}
```

1. Write the Python script

2. Run the start-up command:

- uvicorn main:app --reload
- **Telling uvicorn:** *Run the **app** object defined in the **main.py** file, and keep watching for any file changes so you can auto-reload the server*

Understanding the Start-up Command:

PART	MEANING
uvicorn	The command-line tool to start the Uvicorn ASGI server. It must be installed (pip install uvicorn).
main	Refers to the Python file named main.py (without the .py extension).
app	Refers to the FastAPI application instance in the main.py file. This should be defined as: app = FastAPI().
--reload	Enables auto-reload: the server will automatically restart when you make changes to the code. Useful in development (not recommended in production).

- **Server starts at:** http://127.0.0.1:8000
- **You can visit:**
  - http://127.0.0.1:8000 → Your API endpoint
  - http://127.0.0.1:8000/docs → Swagger UI
  - http://127.0.0.1:8000/redoc → ReDoc docs

Comparative Analysis

16 November 2025 23:42

FEATURE	FastAPI	Flask	Django	Falcon
Release Year	2018	2010	2005	2013
Asynchronous Support	Native (async/await) built-in	Limited (via extensions)	Partial (since Django 3.1)	Full (Not Native)
Performance	Very High (Starlette + async)	Moderate	Moderate	Very High
Type Hinting / Validation	Built-in with Pydantic	Manual	Manual or via DRF	Manual
API Documentation	Auto-generated (Swagger, ReDoc)	No (3rd-party plugins)	Yes (only with DRF)	No
ORM	None built-in (can use SQLAlchemy, Tortoise, etc.)	None built-in	Django ORM	None
Admin Interface	No	No	Built-in	No
Learning Curve	Easy (if familiar with type hints)	Very Easy	Steep (due to monolithic structure)	Medium
Best Use Cases	Modern APIs, ML apps, async microservices	Simple apps, prototyping	Large web apps, admin panels	Ultra-fast microservices, low-latency APIs
Community & Ecosystem	Growing fast	Mature, large	Very mature	Niche, smaller
Built-in Features	Light but powerful	Lightweight	Full-stack batteries-included	Very minimal
Project Structure	Flexible (modular)	Very flexible	Strict	Very flexible
Deployment Ready	(ASGI-ready)	(WSGI)	(WSGI & partial ASGI)	(ASGI/WGSI)

Creating APIs

16 November 2025 19:20

Route/Endpoint in FastAPI:

- A route/endpoint is a specific URL path which our application reaches out to
- Each route is associated with a function (called a ***path operation function***) that executes when that route is accessed

COMPONENT	ROLE
@app.get("/")	Decorator that defines a GET route at root URL
home()	Function that runs when the route is accessed
return	Response is automatically converted to JSON

Returning Responses in FastAPI:

FastAPI allows returning:

- **Dictionaries** (converted to JSON automatically)
- **Pydantic models** (for validation & serialization)
- **Custom response types** (e.g., HTML, plain text, streaming, etc.)

Returning Pydantic Models:

- **When we call:**

```
@app.get("/user", response_model=User)
```

- **What FastAPI does under the hood:**

```
app.add_api_route(
    path="/user",
    endpoint=get_user,
    methods=["GET"],
    response_model=User
)
```

endpoint

path operating function

action

how to validate response data.

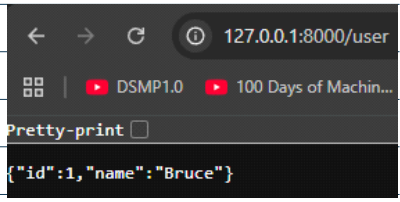
Creating an API which returns a pydantic model

Building APIs > pydantic-demo.py > get_user

```
1   from fastapi import FastAPI
2   from pydantic import BaseModel
3
4   #Define schema of data
5   class User(BaseModel):
6       id: int
7       name: str
8
9   app = FastAPI()
10
11  @app.get("/user", response_model=User)
12  def get_user():
13      return User(id=1, name="Bruce")
14
```

pydantic model

type & data getting returned



HTTP Methods in FastAPI:

- FastAPI supports all standard HTTP methods
- Each method has a semantic purpose in RESTful API design

HTTP METHOD	PURPOSE	SYNTAX
GET	Retrieve data	@app.get()
POST	Create new data	@app.post()
PUT	Update or replace data	@app.put()
DELETE	Delete data	@app.delete()

CRUD Operations

16 November 2025 23:43

Create an app using FastAPI to implement CRUD operations on Employees database

Implement endpoints to:

- Show all employees
- Show particular employee
- Add a new employee
- Update an existing employee
- Delete an existing employee

Employee {
id ,
name ,
dept ,
age
}

- models.py → pydantic models
- main.py → api
  - router
  - path operating function

main.py × models.py

03_employee-app > main.py > delete_employee

```python
from fastapi import FastAPI, HTTPException
from models import Employee
from typing import List

# data base
employees_db: List[Employee] = []

app = FastAPI()

# Endpoint to Read all employees
@app.get("/employees", response_model=List[Employee])
def get_employees():
    return employees_db

# Endpoint to Read specific employee
@app.get("/employees/{emp_id}", response_model=Employee)
def get_employee(emp_id: int):
    for index, emp in enumerate(employees_db):
        if emp.id == emp_id:
            return employees_db[index]
    raise HTTPException(status_code=404, detail="Employee Not Found")
```

```python

# Endpoint to Create new employee
@app.post("/employees", response_model=Employee)
def add_employee(new_emp: Employee):
    for employee in employees_db:
        if employee.id == new_emp.id:
            raise HTTPException(status_code=400, detail="Employee with this ID already exists")
    employees_db.append(new_emp)
    return new_emp

# Endpoint to Update employee
@app.put("/update_employee/{emp_id}", response_model=Employee)
def update_employee(emp_id: int, updated_employee: Employee):
    for index, emp in enumerate(employees_db):
        if emp.id == emp_id:
            employees_db[index] = updated_employee
            return updated_employee
    raise HTTPException(status_code=404, detail="Employee Not Found")

# Endpoint to Delete employee
@app.delete("/delete_employee/{emp_id}")
def delete_employee(emp_id: int):
    for index, employee in enumerate(employees_db):
        if employee.id == emp_id:
            del employees_db[index]
            return {'message': 'Employee deleted successfully!'}
    raise HTTPException(status_code=404, detail="Employee Not Found")
```

Why using index?

id 1 2 3
[e1, e2, e3]
ind 0 1 2

index = id - 1 ✗ This fails later

→ Now, delete e2

id 1 3
[e1, e3]
ind 0 1

Now no relationship b/w index & id.

∴ Use index ✓

main.py models.py ×

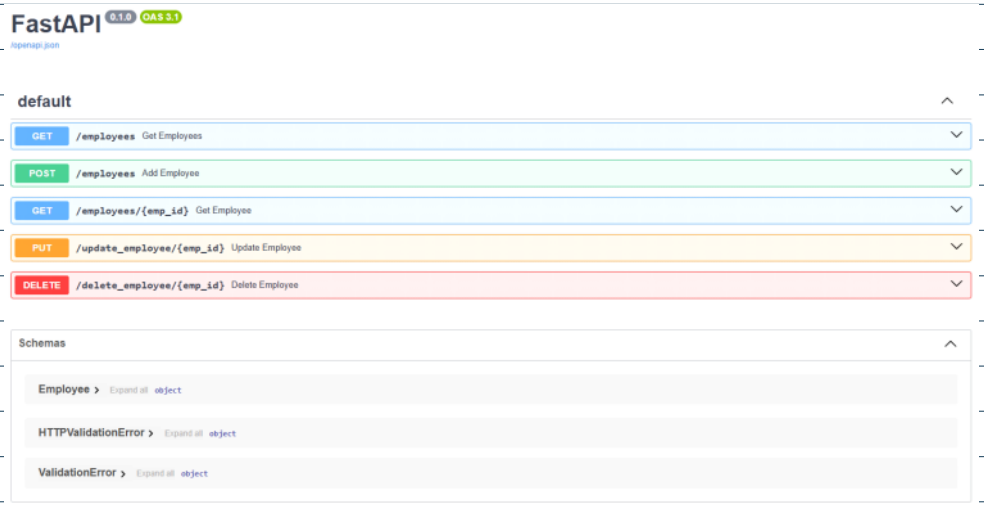
03_employee-app > models.py > Employee

main.py | models.py

03_employee-app > models.py > Employee

```python
#We implement pydantic data models for our employee application.
from pydantic import BaseModel

class Employee(BaseModel):
    id: int
    name: str
    department: str
    age: int
```



Handling Validations and Errors

16 November 2025 23:43

1. Field Validation with Pydantic:

- This can be done using **Field, StrictInt, StrictFloat, etc.**
- In Pydantic, **Field** is used to provide metadata, validations, and default values for fields in a **BaseModel** instance
- Allows for more finer control over input validation and schema generation

Common Parameters of Field:

PARAMETER	DESCRIPTION
default	Default value or ... for required
title	Title for docs/schema
description	Description of the field
example	Example value
gt, ge	Greater than / Greater than or equal (numbers)
lt, le	Less than / Less than or equal (numbers)
min_length	Minimum string length
max_length	Maximum string length
regex	Regex pattern for string validation

main.py | models.py 04_Handling_validation_and_errors U | models-val.py U | models.py 03_employee-app

04_Handling_validation_and_errors > models-val.py > Employee

```python
#We implement pydantic data models for our employee application.
from pydantic import BaseModel, Field

class Employee(BaseModel):
    id: int = Field(..., gt=0, description="The ID must be a positive integer")
    name: str = Field(..., min_length=3, max_length=50)
    department: str = Field(..., min_length=2, max_length=30)
    age: int = Field(..., gt=0)
```

2. Optional Fields & Default Values:

- Use **Optional** from the **typing** module

3. Custom Error Responses:

- Can be implemented using **HTTPException** from **FastAPI**
- Helps indicate the status code along with a custom message

To denote optional values :

To denote optional values :

main.py 03_employee-app | models_val.py U | main.py 04_Handling_validation_and_errors U

04_Handling_validation_and_errors > models_val.py > Employee

```python
#We implement pydantic data models for our employee application.
from pydantic import BaseModel, Field
from typing import Optional

class Employee(BaseModel):
    id: int = Field(..., gt=0, description="The ID must be a positive integer")
    name: str = Field(..., min_length=3, max_length=50)
    department: str = Field(..., min_length=2, max_length=30)
    age: Optional[int] = Field(default = None, gt = 0)

```

It becomes optional field, default = null

pydantic converts "23" to int internally.
but "abc" → error int parsing.

⇒ strict int

main.py 03_employee-app | models_val.py M | main.py 04_Handling_validation_and_errors

04_Handling_validation_and_errors > models_val.py > Employee

```python
#We implement pydantic data models for our employee application.
from pydantic import BaseModel, Field, StrictInt
from typing import Optional

class Employee(BaseModel):
    id: int = Field(..., gt=0, description="The ID must be a positive integer")
    name: str = Field(..., min_length=3, max_length=50)
    department: str = Field(..., min_length=2, max_length=30)
    age: Optional[StrictInt] = Field(default = None, ge = 21)

```

"23" will give error

Asynchronous Programming

16 November 2025 23:43

What is Asynchronous Programming?

- Asynchronous programming is a paradigm that allows your program to perform other tasks while waiting for another operation to complete (ex. database query, API call) to complete, without blocking the execution of the rest of the code
- In other words, instead of waiting for a task to finish before moving on (blocking), you can start a task, and then continue doing other things while that task finishes in the background

Synchronous vs Asynchronous:

OPERATION	SYNCHRONOUS	ASYNCHRONOUS
API Call	Waits for response before proceeding	Sends request, does other work, returns later
Database Query	Blocks until data is fetched	Queries DB, resumes when data arrives
File Read	Waits for disk I/O to complete	Reads in background while other code runs

async and await:

In Python, asynchronous code is written using asyncio module:

- async def Declares an asynchronous function (called a *coroutine*)
- await Tells the interpreter to - *pause here and come back when this operation is done*

Internal Working:

1. Python uses an event loop (via asyncio) to manage asynchronous tasks
2. Tasks are added to the loop
3. When a task hits await, control is yielded back to the loop
4. The loop runs other tasks in the meantime
5. When the awaited task finishes, the loop resumes it

FUNCTION	BLOCKING?	DESCRIPTION
time.sleep(3)	Yes	Pauses the whole thread — blocks everything
await asyncio.sleep(3)	No	Pauses only coroutine — others can run

Use async def if your route does:

- HTTP calls (httpx, aiohttp)
- DB access with async drivers (e.g., asyncpg)
- I/O operations that support async

Use def if:

- Your function is CPU-bound (e.g., ML inference, image processing)
- You're calling a blocking library (e.g., requests, psycopg2)

Eg :

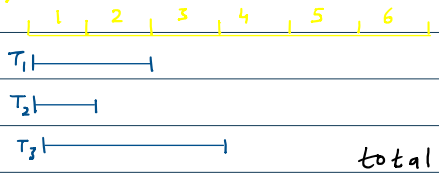
$T_1 \rightarrow 2$ sec

$T_2 \rightarrow 1$ sec

$T_3 \rightarrow 3$ sec

Synch = 2+1+3 = 6 sec

Async :



total time = 3 sec

Database Basics

16 November 2025 19:21

What is a Database?

- A database is a structured collection of data that can be easily accessed, managed, and updated
- In web apps, databases store persistent data (which doesn't get lost)

What are Relational Databases?

These are databases that store data in tables with rows and columns, where each row is a record, and each column is a field

Popular Relational DBs:

DATABASE	FEATURES
SQLite	Lightweight, file-based, no server required. Great for prototyping.
PostgreSQL	Open-source, full-featured, robust. Excellent for production use.
MySQL	Widely used, fast and reliable. Used in many production apps.

Why integrate it with FastAPI?

1. **Persistent Data Storage:**
   - APIs often need to store and persist data across sessions
   - Without a database, all data would live temporarily in memory (RAM) and be lost once the app restarts
2. **Real-World Use Cases:**
   - Most web apps require reliable, structured, and persistent data storage — exactly what databases are built for
3. **Seamless Backend Operations:**
   - Using SQLAlchemy with FastAPI enables users to map these HTTP operations directly to database operations in a clean and Pythonic way
   - Users can implement clean, readable, Python-based models that map to your database
   - It supports asynchronous interactions, aligning perfectly with FastAPI's async-first design
   - Users can utilize FastAPI's dependency injection to cleanly manage DB sessions
4. **Enables Feature-Rich Applications:**
   - With a database in place, your FastAPI application can:-
     - Add authentication and authorization ✓
     - Track historical data ✓

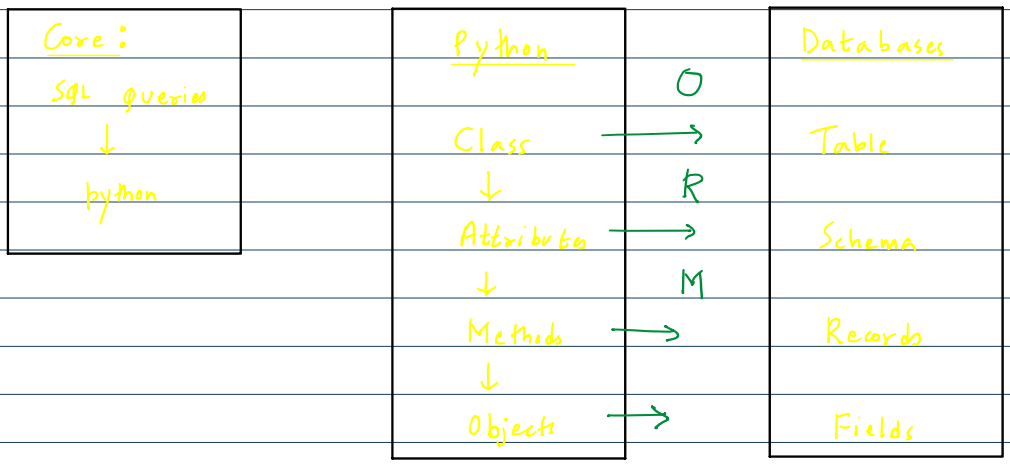
4. **Enables Feature-Rich Applications:**
   - With a database in place, your FastAPI application can:-
     - Add authentication and authorization
     - Track historical data
     - Perform analytics
     - Manage file uploads and metadata
5. **Security and Data Integrity:**
   - Relational databases provide:
     - Data integrity constraints
     - Transactions to ensure consistency
     - Access control and permissions
   - This is essential for building secure and reliable APIs

SQLAlchemy Basics

18 November 2025 00:44

1. What is SQLAlchemy ?

- SQLAlchemy is a powerful Python SQL toolkit and Object Relational Mapper (ORM) that helps Python applications interact with relational databases
- SQLAlchemy Has Two Main Parts:
  - **SQLAlchemy Core:**
    - A lower-level SQL expression language that lets users build SQL queries using Python
    - Involves writing raw SQL, but in Pythonic syntax
  - **SQLAlchemy ORM (Object Relational Mapper):**
    - A higher-level tool that lets users map Python classes to database tables
    - Users write Python code, and SQLAlchemy handles the SQL under the hood



2. Why Use SQLAlchemy?

FEATURE	DESCRIPTION
ORM	Easily map Python classes to database tables
Cross-DB Compatibility	Works with PostgreSQL, MySQL, SQLite, etc.
Security	Protects against SQL injection
Asynchronous Support	Plays well with async in FastAPI and modern Python
Mature & Well-Documented	Battle-tested and production-ready

3. Installation:

- pip install sqlalchemy sqlalchemy[asyncio]

CRUD App Project

18 November 2025 01:23

This app is a simple REST API to manage employee records, implementing CRUD operations using:

- **FastAPI** — Web framework for building APIs
- **SQLAlchemy** — ORM (Object Relational Mapper) for database interaction
- **SQLite** — Lightweight database for storage
- **Pydantic** — Data validation and serialization

App Structure:

- **crud-app**
  - database.py
  - models.py
  - schemas.py
  - crud.py
  - main.py

Model Serialization

16 November 2025 19:21

What is Model Serialization?

- Serialization is the process of converting a trained machine learning model into a byte stream that can be saved to a file or database
- This can later be deserialized (loaded) to recreate the model in memory without retraining it from scratch

Common Libraries:

- Pickle
- Joblib

] for python obj / ML model

- Keras (.h5, .keras)

] for keras model

- Tensorflow (SavedModel)
- Pytorch (.pt, .pth)

Common Formats:

- JSON
- Binary
- HDF5

Why Model Serialization is Important:

1. **Saves Time and Computational Resources:**
   - Training ML models, especially deep learning models, can take minutes to hours—or even days
   - Serialization allows you to store the trained model once and use it repeatedly without incurring the cost of retraining
   - This is especially critical in:
     - Production deployments
     - Iterative testing
     - Rapid prototyping
     - Resource-constrained environments (like edge devices)
2. **Portability Across Platforms and Environments:**
   - A serialized model can be moved across different machines, operating systems, or cloud services
   - It facilitates collaboration across teams—one team trains the model, another team deploys it
   - Serialization also allows ML models to be integrated into mobile apps, IoT devices, or cloud containers (ex. Docker)
3. **Reproducibility and Consistency:**
   - Serialization preserves the trained state exactly—including model parameters, weights, and internal

configurations

- Ensures consistent predictions across different runs, which is crucial for:
  - Model validation
  - A/B testing
  - Regulatory or compliance audits

4. **Model Serving and Integration:**
   - Serialization enables seamless deployment of ML models into real-world systems like REST APIs, web applications, dashboards, or edge devices
   - Allows decoupling of training and inference phases:
     - Training happens offline (e.g., Jupyter notebook)
     - Inference happens online (e.g., real-time request to a FastAPI or Flask server)
5. **Foundation for Model Versioning and CI/CD Pipelines:**
   - Serialization plays a vital role in MLOps:
     - Store models in versioned model registries (e.g., **MLflow**, **DVC**, **AWS SageMaker**)
     - Automate model deployment and rollback
   - Helps teams track changes and compare performance across versions

Pickle vs Joblib

21 November 2025 21:17

FEATURE	pickle	joblib
Purpose	General-purpose serialization	Optimized for objects with large NumPy arrays (common in ML)
Library	Built-in (import pickle)	External (pip install joblib)
Serialization Format	Binary	Binary (optimized with efficient NumPy storage)
Performance	Slower for large NumPy arrays	Faster for large NumPy arrays due to efficient memory mapping
Use Case	Any Python object	Large data like ML models, NumPy arrays, and SciPy objects
Compression Support	Manual (you must use gzip, bz2, etc. separately)	Built-in (compress=True)
Parallel I/O	Not supported	Supported (internally uses multiple I/O operations)
File Size	Larger with numerical arrays	Smaller for numerical data due to compression
Backward Compatibility	Good	Similar to pickle, but better compatibility with NumPy
Common ML Usage	Small models (e.g., decision trees, dicts)	Large models (e.g., scikit-learn pipelines with arrays)

Pickle

write binary

```
[21] ✓ 0s
#Serialization
with open('model.pkl', 'wb') as f:
  pickle.dump(model, f)
```

read binary

```
[22] ✓ 0s
#Deserialization (loading the model)
with open('model.pkl', 'rb') as f:
  pickle_model = pickle.load(f)
```

```
[23] ✓ 0s
pickle_model
```

```
LinearRegression
LinearRegression()
```

```
[24] ✓ 0s
pickle_model.predict(X)
```

```
array([411270.20746023, 415943.95527539, 380534.16332108, ...,
       24989.18869283,  37961.51486098,  55555.76850636])
```

Joblib

```
[27] ✓ 0s
#Serialization
joblib.dump(model, 'model.joblib')
```

```
['model.joblib']
```

```
[30] ✓ 0s
#Deserialization
joblib_model = joblib.load('model.joblib')
joblib_model
```

```
LinearRegression
LinearRegression()
```

```
[32] ✓ 0s
y_pred_joblib = joblib_model.predict(X)
y_pred_joblib
```

```
array([411270.20746023, 415943.95527539, 380534.16332108, ...,
       24989.18869283,  37961.51486098,  55555.76850636])
```

```
[33] ✓ 0s
r2_score(y, y_pred_joblib)
```

```
0.6369116857335635
```

Keras

Exporting the model

[33]
✓ 0s

```
model.save('keras_model.keras')
```

Importing (Load) Model

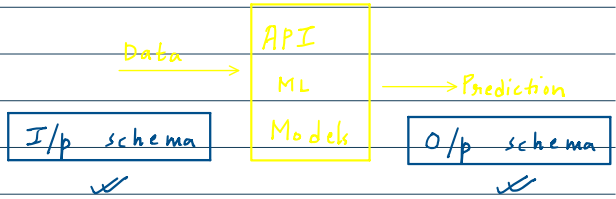
[34]
✓ 0s

```
keras_model = load_model('keras_model.keras')
```

Designing Model I/O Schemas

21 November 2025 21:17

Why Define Input and Output Schemas?



1. **Data Validation and Type Safety:**
   - Without schema validation, you're blindly trusting that the incoming data is well-formed—which is risky
   - Can implement automatic type checking using Pydantic and rules using Field
2. **Clear API Contracts:**
   - Schemas define a contract for how your API should be used
     - InputSchema = what users should send
     - OutputSchema = what your app will return
3. **Improved Developer Experience:**
   - FastAPI auto-generates beautiful interactive docs (Swagger UI)
   - Built-in validation error messages help frontend/backend developers debug easily
   - Less guesswork = faster development and fewer bugs
4. **Cleaner Code and Reusability**
5. **Secure and Robust APIs**
6. **Managing Nested & Complex Structures:**
   - ML applications often involve:
     - Nested JSON structures
     - Optional fields
     - Lists of structured items
   - Schemas make these easy to define and validate using **BaseModel**, **Optional**, **List**, etc.
7. **Logging, Auditing, and Testing:**
   - Well-defined schemas simplify structured logging
   - Makes it easier to write tests with known input/output formats
   - Help trace issues back to specific schema validation failures

Serving ML Models

21 November 2025 21:17

Middleware

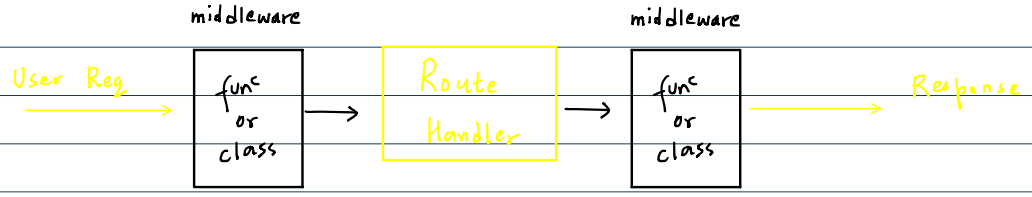
16 November 2025 19:21

What is a Middleware?

- Middleware is a function or class that intercepts incoming requests before they reach the route handler, or outgoing responses after the route handler has processed the request
- Middleware is a function that runs before or after each request in the application
- Middleware allows us to:
  - Log requests and responses
  - Handle CORS or custom headers
  - Measure performance
  - Catch and process errors globally

```
@app.get('/')
def index():
{
}
```

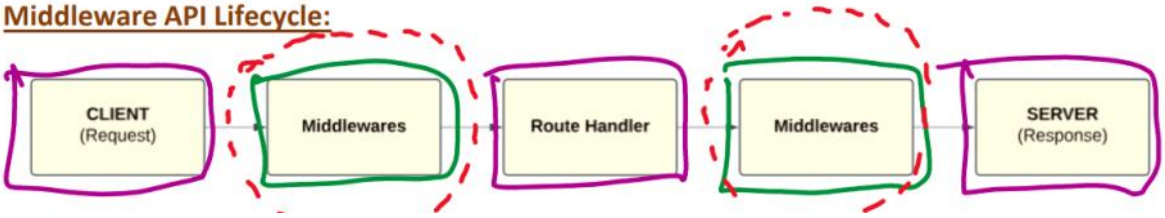
this function is route handler



Main Idea Behind Middleware:

- FastAPI uses the concept of middleware similar to other modern frameworks like Node.js or Django
- Middleware runs in a chain, one after another, in the order they are registered
- Middleware can modify:
  - The request before it's passed to the endpoint
  - The response returned by the endpoint

Middleware API Lifecycle:



Dependency Injection

25 November 2025 00:53

What is Dependency Injection?

- Dependency Injection (DI) is a software design pattern that allows objects or functions to receive their dependencies from external sources rather than creating them internally
- FastAPI uses the Depends class to resolve and inject dependencies automatically

Common Use Cases of Dependency Injection:

1. Database Connections
2. Configuration Management
3. User Authentication
4. Background Task Setup

Best Practices:

PRACTICE	DESCRIPTION
Keep dependencies pure	No side effects; easier to test and reuse
Use classes for related parameters	Better structure and organization
Avoid heavy computation inside dependencies	Keep them fast and efficient
Use yield for resource management	Useful for DB sessions, file access, etc.
Group reusable logic	Like authentication, configuration, logging
Apply dependencies at router/middleware level	For authentication, rate limiting, etc.
Override dependencies in tests	Isolate logic and improve test reliability

JWT Authentication

25 November 2025 00:54

What is JWT?

- JWT (**JSON Web Token**) is a compact, URL-safe means of representing claims between two parties
- JWTs allow to **validate user identity** and **protect secure routes** without needing to store session state on the server
- Commonly used for authentication and information exchange ✓

JWT Structure:

- **Header** – Metadata (e.g., algorithm used)
- **Payload** – Claims like user ID, expiration time, etc.
- **Signature** – Ensures token integrity using secret key

Installation: Install dependencies required for secure token handling and password hashing

```
pip install fastapi uvicorn authlib passlib[bcrypt]
```

- **authlib:**
  - Used for building **authentication and authorization systems**
  - Implements modern security protocols like **OAuth2** and **JWT** properly
  - Eliminates the need to combine multiple packages
  - Smooth integration with FastAPI, Flask, etc.
- **passlib[bcrypt]**
  - Is a password hashing library
  - Mainly used for securely hashing and verifying passwords
  - **[bcrypt]** installs **bcrypt**, a secure hashing algorithm, used for storing passwords in a safe way

Implementation of JWT Authentication for Login in FastAPI:

- **auth.py**
- **models.py**
- **utils.py**
- **main.py**

Managing API Keys

25 November 2025 00:55

What is an API Key?

- An API key is like a password for accessing a web service
- It's a long string of letters and numbers that:
  - Identifies who is making the request
  - Verifies whether they have permission to access the data/service
  - Helps the API provider track usage

Why Do We Need API Keys?

1. **Authentication & Authorization:**
   - To verify that a request is coming from a trusted source
   - Some APIs have public access, but many restrict data to **authenticated users**
2. **Rate Limiting:**
   - Prevents abuse by limiting how many requests a single user or app can make in a given time
   - Without API keys, it's hard to control **spam or overload**
3. **Usage Tracking:**
   - Helps the API provider analyze how their service is being used
   - Enables billing or quota enforcement for paid APIs
4. **Security:**
   - Prevents unauthorized access to sensitive data or actions
   - Can be revoked or rotated if compromised

Why Do We Need API Keys?

Misbah

1. **Authentication & Authorization:**
   - To verify that a request is coming from a trusted source
   - Some APIs have public access, but many restrict data to **authenticated users**

2. **Rate Limiting:**
   - Prevents abuse by limiting how many requests a single user or app can make in a given time
   - Without API keys, it's hard to control **spam or overload**

3. **Usage Tracking:**
   - Helps the API provider analyze how their service is being used
   - Enables billing or quota enforcement for paid APIs

Implementing API Keys with FastAPI via:

Two ways :-

- **Headers**
- **Environment Variables (.env)**

Best Practices

25 November 2025 00:55

- Use **HTTPS** in production (Middleware)
- Store **JWT** secrets and **API keys** in **.env** file or **secure vaults**
- Set proper **CORS** and **CSRF** protections:
  - **CORS (Cross-Origin Resource Sharing):**
    - Security feature implemented by web browsers to control how web pages from one origin (domain) can request resources from another origin
  - **CSRF (Cross-Site Request Forgery):**
    - Web security vulnerability where a malicious website tricks a user's browser into performing unwanted actions on a different website where the user is authenticated
- Use hashing (e.g., **bcrypt**) for passwords
- Set token expiration time
- Implement **Role-Based Access Control** (**RBAC**) where needed

Importance of Testing APIs

16 November 2025 19:21

- Testing is a non-negotiable aspect of modern API development
- It ensures that the web application functions correctly, is reliable, and behaves consistently even as you scale the application

1. **Correctness of Application Logic:**
   - Verifies that the API behaves as expected under different conditions
   - Reduces bugs in production and catches logical errors early
   - Ensures ML models make correct predictions and sensitive endpoints do not expose data due to logic errors

2. **Protection Against Regressions:**
   - Regressions refer to the bugs introduced into previously working code when new changes are made
   - Helps maintain backward compatibility and stability of endpoints
   - Use regression test suites to validate critical user flows like authentication, payments, data submissions, etc.

3. **Safety Net During Refactoring:** (Modifying code)
   - Helps improve code readability, performance, and maintainability
   - Automated test cases verify that the refactored code still behaves the same way as before
   - Run the test suite after every major code change or cleanup

4. **Enables Continuous Integration Pipelines:** (CI)
   - Tests act as a gatekeeper to production; code won't be merged unless all tests pass
   - Ensures stable deployments, boosts team confidence in merging PRs and reduces manual testing effort

BENEFIT	DESCRIPTION
Correctness	Catches bugs before they reach users
Regression Safety	Prevents breaking existing functionality
Refactor Confidence	Enables cleaner, better code
CI Integration	Automates quality checks and speeds up delivery

Mock ML Models

13 December 2025 15:54

When we test an API with ML models,
instead of using the real model we will replace it with a mock object.

What is meant by Mocking ML Models?

- Mocking an ML model means replacing the actual ML model with a fake (or simulated) object in your tests that behaves like the real model but doesn't perform any actual computation
- Instead of loading a large model file and calling its predict() method, you pretend (mock) that a model exists and will return a known, controlled value

1. **ML Models Are Heavy to Load:**
   - In production-grade ML APIs, models may be trained on large datasets and saved as serialized **.pkl**, **.joblib**, or **.h5** files
   - These files can be hundreds of MBs or even GBs in size and may contain complex architectures (e.g., deep learning models)
   - Deserializing and initializing these models can take several seconds or even minutes
   - Running this load operation for every test case makes the test slow, resource-intensive, and prone to timeouts or memory exhaustion—**not ideal for a CI/CD pipeline**

2. **Tests Should Focus on API Logic:**
   - When testing an API endpoint (e.g., **/predict**), the goal is to check if the endpoint accepts input and returns a response in the correct format
   - We do not care about whether the prediction itself is accurate
   - Hence, using a real model introduces unnecessary complexity and violates unit testing principles

3. **Mocking Makes Tests Fast and Deterministic:**
   - Mocking replaces the actual model object with a fake object that simulates the **predict** method
   - This fake object can return predefined outputs for known inputs, simulate exceptions to test error handling and void real computation, making the test lightweight and fast
   - As a result, test becomes deterministic; the same input always yields the same response, which helps maintain stability in test results

BENEFIT	DESCRIPTION
Faster Execution	No time spent loading model files or running computation-heavy predictions
Better Isolation	Focus on testing API routes and business logic, not ML internals
Error Simulation	Easily test how your app handles errors like model.predict() throwing an exception
CI/CD Friendly	Lightweight tests that can run on every commit in seconds
Deterministic Results	Avoid randomness introduced by some models (e.g., unseeded probabilistic models)

Common API Errors

13 December 2025 15:54

1. 401 Unauthorized:

- **Meaning:** Authentication has failed — the server didn't get a valid token or credentials
- **Common Causes:**
  - Missing or expired authentication token
  - Wrong API key or credentials
  - Bearer token not included or formatted incorrectly
- **Debugging Tips:**
  - Confirm if the token is valid and not expired
  - Ensure the **Authorization** header is set properly
  - Double-check API key/token permissions

2. 404 Not Found:

- **Meaning:** The URL or resource requested does not exist on the server
- **Common Causes:**
  - Typo in endpoint URL
  - The route isn't defined on the backend
  - Missing trailing slash (for some frameworks like Django REST Framework)
- **Debugging Tips:**
  - Recheck the API path spelling and method (GET/POST/etc.)
  - Confirm if the endpoint exists in the backend routing logic
  - Use API documentation or tools like **Swagger/OpenAPI** to test routes

3. 422 Unprocessable Entity:

- **Meaning:** The server understands the request, but it can't process the data because it doesn't match the expected format or required fields are missing
- **Common Causes:**
  - Missing required fields in the request body
  - Wrong data types (e.g., sending a string instead of an integer)
  - Failing validation checks (e.g., string too short, email not valid)
- **Debugging Tips:**
  - Check the API documentation/schema carefully
  - Validate the payload locally using a schema validator (ex. Pydantic)
  - Use tools like **Postman** or **curl** to test with valid input

4. 500 Internal Server Error:

- **Meaning:** Something went wrong on the server side while processing the request
- **Common Causes:**
  - Unhandled exception in backend code (e.g., division by zero, missing ML model)
  - Database connection issues
  - Misconfiguration in server code or environment
- **Debugging Tips:**
  - Check server logs for traceback or error stack
  - Add proper exception handling (try-except blocks)
  - Ensure all dependencies and models are loaded before request handling
  - Use **logging** to catch unexpected exceptions

Debugging Techniques

13 December 2025 15:54

FastAPI makes Debugging easier through:

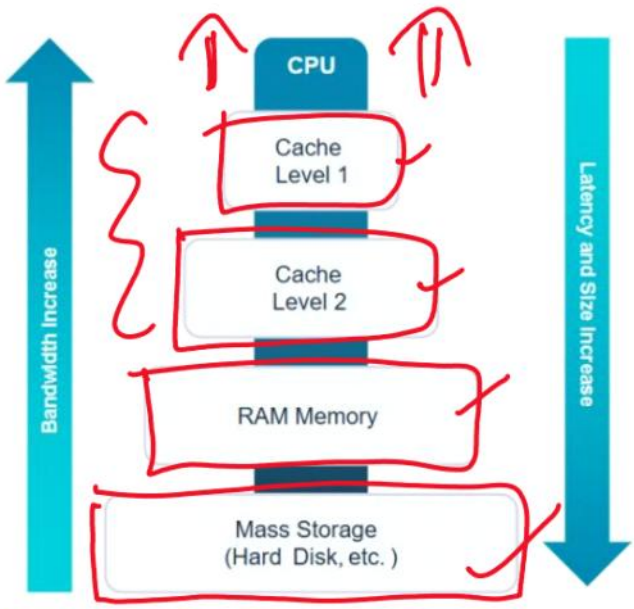
- Structured Logging
- Exception Handling
- API testing tools like Postman/curl
- Development Mode Configurations

1. Caching

16 November 2025 19:21

What is Caching?

- Caching is the process of storing a copy of data or computational results in a temporary storage layer (**cache**)
- This is done so that future requests for that same data can be served much faster, without needing to recompute or fetch it from the original source



An Overview of Memory Caching

2. Caching with Redis

23 December 2025 16:30

What is Redis?

- Redis (short for REmote DIctionary Server) is a fast, open-source, in-memory data structure store
- Redis stores everything in memory, which allows for blazing fast reads and writes — **often under 1 millisecond latency**
- **Redis is commonly used as:**
  - **Key-Value Database:** Stores data as key-value pairs, similar to a Python dictionary
  - **Cache:** Frequently used to cache database queries, API responses, and ML model predictions
  - **Message Broker:** Supports **publish/subscribe (pub/sub)**, streams, and queues for building messaging systems

- It does not use any physical memory.

Why is Redis fast?

- **In-memory:** No disk I/O during reads/writes
- **Single-threaded:** Avoids context switching and locking overhead
- **Optimized C codebase**

Redis Persistence: (Retaining data)

Though Redis is an in-memory store, it supports data persistence through:

- **RDB (Redis Database Backups)** – snapshots at intervals
- **AOF (Append Only File)** – logs every write operation

3. Redis with FastAPI

23 December 2025 16:30

4. Profiling FastAPI apps

23 December 2025 16:30

Why Profiling is Important:

- Identify performance bottlenecks in business logic or API calls
- Minimize latency and maximize throughput
- Reduce CPU, memory usage, and I/O wait times
- Make informed architectural decisions (e.g., sync vs async)
- Prepare APIs for scale

Understanding behaviour and patterns of APIs

Key Metrics to Observe:

METRIC	DESCRIPTION	
Response Time	Total time to complete a request (latency)	lower better
CPU Usage	How much CPU is consumed during execution	lower better
Memory Usage	RAM consumed by the app or request	lower better
Throughput	Number of requests served per second	higher better
Error Rate	Percentage of failed requests under load	lower better

Profiling Tools:

- **time:** Quick & dirty timing of functions (Quick & simple)
- **cProfile:** Built-in profiler to capture function calls & time
- **line_profiler:** Line-by-line profiling (detailed profiling)

5. Benchmarking APIs

23 December 2025 16:31

What is Benchmarking of APIs?

- **Benchmarking** is the process of **measuring the performance** of an application or system under specific conditions
- In the case of APIs, it provides **quantitative insights** into how they behave under different levels of usage, helping make **data-driven** decisions for improvement

6. Monitoring APIs

23 December 2025 16:31

What is API Monitoring?

- Monitoring APIs refers to the continuous observation and tracking of an API's performance, availability, and functionality to ensure it behaves as expected and delivers a seamless experience to users or connected systems
- It is a critical aspect of maintaining reliable and high-performing software applications, especially in production environments

Why API Monitoring?

1. **Availability Monitoring:**
   - Ensures the API is reachable and responsive
   - Uptime checks at regular intervals
   - Alerts when the API is down or returns errors
2. **Performance Monitoring:**
   - Measures response time, latency, throughput
   - Identifies slow endpoints or inefficient logic
   - Helps in tuning APIs for better user experience
3. **Error Tracking:**
   - Logs and analyzes status codes
   - Tracks frequency and type of errors
   - Useful for debugging and **root cause analysis (RCA)**
4. **Usage Analytics:**
   - Monitors request volume, endpoints usage, and consumer behavior
   - Helps in capacity planning and scaling
5. **Resource Monitoring:**
   - Tracks CPU, memory, I/O utilization at the infrastructure level
   - Useful in understanding backend stress caused by API calls
6. **Alerting & Incident Management:**
   - Sends real-time notifications on abnormal behavior
   - Helps reduce **Mean Time To Detect (MTTD)** and **Mean Time To Resolve (MTTR)**

Key Metrics to Track: Similar to Benchmarking

6.1 - Prometheus

23 December 2025 16:37

What is Prometheus?

- Prometheus is an open-source systems monitoring and alerting toolkit, originally developed by **SoundCloud**
- It's designed for reliability and scalability, and is especially strong in environments like cloud-native applications, microservices, and containerized deployments (like **Kubernetes**)

Characteristics:

1. **Pull-based Model:** Prometheus **pulls (scrapes)** metrics from instrumented targets (applications/services) at specified intervals
2. **Time Series Storage:** Metrics are stored as **time series** and they're indexed by :-
   - A metric name
   - One or more labels
3. **Flexible Query Language:** **PromQL** lets you perform complex queries to **aggregate, filter, and compute metrics**
4. **Built-in Alert Manager:** Allows to configure alert rules and send alerts to email, Slack, PagerDuty, etc.

How Prometheus scrapes Metrics?

1. **Configuration:** Prometheus reads a **YAML** configuration file (**prometheus.yml**) to determine what to scrape, when, and how often
2. **Targets:** Each target is an HTTP endpoint that exposes metrics, commonly at the **/metrics** route
3. **Scrape Format:** The data should be exposed in **Prometheus text** format or **OpenMetrics** format

Use Cases:

- Monitoring web services
- Observing containerized applications
- Tracking application health, traffic, and performance
- Generating real-time alerts and dashboards

6.2 - Prometheus with FastAPI

23 December 2025 16:37

prometheus-fastapi-instrumentator:

- **prometheus-fastapi-instrumentator** is a plug-and-play library that enables **automatic instrumentation** of FastAPI applications
- It helps collect **runtime metrics** (request count, latency and status codes) and expose them in a **Prometheus-compatible** format, typically at **/metrics**
- **Installation:**
  - **pip install prometheus-fastapi-instrumentator** (Python 3.9+)
  - **pip install prometheus-fastapi-instrumentator==5.9.1** (up to Python 3.8)

FastAPI Integration: http://127.0.0.1:8000/metrics

- **instrument(app):**
  - Hooks into FastAPI's **routing layer**
  - Captures HTTP-level metrics:
    - Request count
    - Request duration
    - Status code distribution
    - Method and endpoint path
  - Wraps every route handler with logic to collect and export the metrics
- **expose(app):**
  - Adds a **/metrics** route (by default)
  - This endpoint serves metrics in a **Prometheus-compatible** format

Default Metrics Generated:

- **method:** HTTP method (GET, POST, etc.)
- **path:** API route
- **status:** HTTP status code
- **_bucket{le="0.1"}:** Histogram bucket showing how many requests completed in ≤0.1 seconds
- **_count** and **_sum:** Total number and cumulative duration of requests

(le ⇒ less than equal to)

6.3 - Prometheus, FastAPI & Docker

23 December 2025 16:37

File Structure:

project-folder/

- app/
  - main.py
- prometheus/
  - prometheus.yml
- docker-compose.yml
- Dockerfile

Run the app: docker-compose up --build

Access the Interfaces:

- **FastAPI endpoint:** http://localhost:8000/
- **FastAPI metrics:** http://localhost:8000/metrics
- **Prometheus UI:** http://localhost:9090/
  - **Run query:** **http_requests_total**

http://localhost:8000
http://localhost:8000/metrics
http://localhost:9090

Docker-compose:

- Higher-level tool used to define and run **multi-container** Docker applications
- Uses a **YAML file (docker-compose.yml)** to configure application services, networks, volumes, etc.
- Allows to start all defined services with one command: **docker-compose up**
- **Key Benefits:**
  - Simplifies running multi-container environments
  - Handles networking and volume mounts between containers automatically

FEATURE	DOCKER	DOCKER COMPOSE
Scope	Single container	Multi-container applications
Configuration	Command-line options	YAML configuration (docker-compose.yml)
Networking	Manual or implicit	Automatically shared network across services
Use Case	Simple apps or dev/test containers	Complex environments (e.g., microservices)
Command Examples	**docker run, docker build**	**docker-compose up, docker-compose down**

6.4 - Grafana

23 December 2025 16:38

What is Grafana?

- Grafana is an open-source analytics and interactive data visualization platform
- It allows users to query, visualize, alert on, and understand metrics no matter where they are stored
- It is widely used for monitoring infrastructure, applications, and data pipelines in real time
- It supports pluggable data sources, meaning it can connect to a wide range of backends like Prometheus, InfluxDB, Elasticsearch, MySQL, PostgreSQL, and many more

How Grafana Works:

1. **Data Sources:**
   - Grafana connects to databases and time-series backends via **data source plugins**
   - Common sources include: **Prometheus, InfluxDB, Loki, Elasticsearch, PostgreSQL, MySQL**
2. **Queries:**
   - You can write **custom queries** or use **built-in query builders** to extract data from the source
3. **Dashboards:**
   - Dashboards are made of **panels** (charts, tables, gauges, heatmaps, etc.) where the data is visualized
   - You can save and share dashboards with teams
4. **Alerting:**
   - Grafana provides **rule-based alerting** on your metrics
   - You can integrate with **Slack, PagerDuty, Microsoft Teams, Email**, etc., for notifications

Use Cases of Grafana:

1. **Infrastructure Monitoring:**
   - Monitor server CPU, RAM, disk usage, network I/O
   - Integrate with **Prometheus, Telegraf, InfluxDB**, or **Node Exporter**
2. **Application Performance Monitoring (APM):**
   - Track API response times, throughput, error rates
   - Useful with **Prometheus + FastAPI/Django**, or APM tools like **Jaeger**
3. **Database Monitoring:**
   - Connect to **MySQL, PostgreSQL, MongoDB**, etc., and visualize query performance, connections, latency
4. **DevOps and CI/CD Pipeline Monitoring:**
   - Visualize deployment frequency, failure rates, build times using data from **Jenkins, GitHub Actions, CircleCI**, etc.

5. **IoT and Sensor Data Dashboards:**
   - Ingest data from IoT sensors using **MQTT** or **InfluxDB**, visualize in Grafana with time-series plots

6. **Business KPIs Monitoring:**
   - Track sales performance, user engagement, or churn rates using **Google Sheets, PostgreSQL,** or **Excel files** as data sources

7. **Security Monitoring:**
   - Visualize login attempts, suspicious activity using **Elasticsearch** or **Loki** for log aggregation

Advantages of Grafana:

- **Open-source and free:** Core Grafana is free to use and has a large community of contributors
- **Custom Dashboards:** Easy to build and customize dashboards tailored to specific needs
- **Plugin Ecosystem:** Rich library of community-developed plugins for new data sources or visuals
- **Multiple Data Sources:** Combine metrics from multiple tools in a **single unified dashboard**
- **Time-Series Friendly:** Especially powerful for time-series and real-time streaming data
- **Integrations:** Works with **Prometheus, Loki, Elasticsearch, InfluxDB, AWS CloudWatch**, etc.
- **Alerting System:** Allows users to receive notifications when metrics cross thresholds
- **Collaboration:** Share dashboards with teams and define permissions for access control
- **Templating:** Use variables to create dynamic and reusable dashboards

6.5 - Grafana, Prometheus, FastAPI & Docker

23 December 2025 16:38

File Structure:

```
project-folder/
  - app/
      - main.py
      - Dockerfile
  - prometheus/
      - prometheus.yml
  - docker-compose.yml
  - requirements.txt
```

Run the app: `docker-compose up --build`

Access the Interfaces:

- **FastAPI endpoint:** http://localhost:8000/
- **FastAPI metrics:** http://localhost:8000/metrics
- **Prometheus UI:** http://localhost:9090/
  - **Run query:** `http_requests_total`
- **Grafana:** http://localhost:3000
  - **Username:** admin
  - **Password:** admin

http://localhost:8000/

http://localhost:8000/metrics

http://localhost:9090/

http_requests_total

http://localhost:3000

Configure Grafana:

- Navigate to **Grafana -> Add data source -> Prometheus**
- **Set URL:** http://prometheus:9090
- Save & Test
- Create dashboards using metrics such as:
  - **http_server_requests_total**
  - **http_request_duration_seconds_bucket**
  - **http_request_duration_seconds_sum**

http://prometheus:9090

1. Project Description & Dataset

16 November 2025 19:21

1. Project Description & Dataset

Saturday, July 5, 2025 12:54 PM

Car Price Prediction using FastAPI with deployment over Render and Redis

Motivation:

- To tie together all concepts learned so far
- Implement an end-to-end project with deployment
- Follow industry-level best practices

Dataset:

- 16 features
- 8 categorical features
- 8 numerical features
- Includes missing values

2. Project Structure

26 December 2025 00:49

project-folder/

- **app/**: Main application package containing all FastAPI app components
  - **__init__.py**: Initializes the Python package for **app**
  - **main.py**: sets up routes, middleware, and monitoring
  - **models/**
    - **model.joblib:** Serialized ML model used for prediction
  - **api/:** Contains route definitions for API endpoints
    - **__init__.py**
    - **routes_predict.py:** Defines the **/predict** route for price predictions
    - **routes_auth.py:** Defines the **/login** route for user authentication via JWT
  - **core/:** logic for config, security, dependencies, and exception handling
    - **config.py:** Loads environment variables and app-wide settings
    - **security.py:** Handles JWT creation and verification logic
    - **dependencies.py:** Dependency injection logic for API key and JWT token validation
    - **exceptions.py:** Custom exception handlers for consistent error responses
  - **services/**
    - **model_service.py:** Loads the ML model and performs predictions (with Redis caching)
  - **middleware/**
    - **logging_middleware.py:** Logs all incoming requests and outgoing responses
  - **cache/:**
    - **redis_cache.py**
  - **utils/:** Utility modules for common functionality
    - **logger.py:** Custom logger configuration (optional)
- **notebooks/:** Jupyter notebooks for experimentation
- **data/:**
- **training/:**
  - **__init__.py:**
  - **train_utils.py:** common functions to support model training
  - **train_model.py:** model training script
- **requirements.txt:** List of Python dependencies required for the app

    - **__init__.py:**
    - **train_utils.py:** common functions to support model training
    - **train_model.py:** model training script
- **requirements.txt:** List of Python dependencies required for the app
- **Dockerfile:** Docker image definition to containerize the FastAPI app
- **docker-compose.yml:** Orchestrates FastAPI, Redis, Prometheus, and Grafana services
- **prometheus.yml:** Monitor FastAPI app using the Prometheus FastAPI Instrumentator
- **render.yaml:** Automates deployment settings of web services over Render
- **.env:** Environment variables used by the application
- **README.md:** Project overview, setup instructions, and usage guide

3. Project Setup and Files

26 December 2025 00:49

3. Project Setup with GitHub

Saturday, July 5, 2025 1:27 PM

- Create a remote repository on GitHub
- Initialize local git repository
- Connect local and remote repositories
- Make initial commit

4. Configurations and Security

Saturday, July 5, 2025 1:54 PM

Files to setup:

- .env
- app/core/config.py
- app/core/security.py

5. Auth & Dependencies

Saturday, July 5, 2025 2:06 PM

Files to setup:

- app/core/dependencies.py
- app/core/exceptions.py
- app/api/routes_auth.py

6. ML Integration with Caching

Saturday, July 5, 2025 2:13 PM

Files to setup:

- notebooks/sample.ipynb
- training/train_utils.py
- training/train_model.py
- app/cache/redis_cache.py
- app/services/model_service.py
- app/api/routes_predict.py

7. Middlewares and API

Sunday, July 6, 2025 12:04 PM

Files to setup:

- app/middleware/logging_middleware.py
- app/main.py

8. Monitoring & Containerization

Sunday, July 6, 2025 12:11 PM

Files to setup:

- prometheus.yml
- Dockerfile
- docker-compose.yml

9. Running Locally

26 December 2025 00:49

Files to setup:

- requirements.txt

Run the application:

- docker-compose up --build

docker-compose up --build

Interfaces:

- FastAPI endpoint: http://localhost:8000/
- FastAPI metrics: http://localhost:8000/metrics
- Prometheus UI: http://localhost:9090/
  - Run query: http_requests_total
- Grafana: http://localhost:3000
  - Username: admin
  - Password: admin

http://localhost:8000/
http://localhost:8000/metrics
http://localhost:9090/

http_requests_total

http://localhost:3000

Configure Grafana:

- Navigate to Grafana -> Add data source -> Prometheus
- Set URL: http://prometheus:9090
- Save & Test
- Create dashboards using metrics such as:
  - http_server_requests_total
  - http_request_duration_seconds_bucket
  - http_request_duration_seconds_sum

http://prometheus:9090

10. Deployment

26 December 2025 00:49

10. Deployment

Sunday, July 6, 2025 2:08 PM

Files to setup:

- **render.yaml**

Steps:

- **Visit:** https://redis.com/try-free/
- **Sign-in with GitHub/Email**
- **Create a new free database**
- **Copy Redis URL**
- **Update redis_cache.py**
- **Visit:** https://render.com/
- **Sign-in with GitHub**
- **Add new Web Service**
- **Select repository**
- **Deploy Web Service**