1. Contents

Tuesday, January 14, 2025 1:35 PM

2. Prerequisites

Tuesday, January 14, 2025 1:38 PM

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

1. Definition

Sunday, January 19, 2025 7:51 AM

- 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

3. Types of APIs

Sunday, January 19, 2025 8:05 AM

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

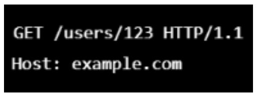
- Provide an abstraction layer to control and retrieve data from devices without needing low-level programming
- Examples:
  - *GPU APIs like CUDA or OpenCL for performing parallel computations*
  - *APIs for IoT devices such as sensors or smart appliances*
  - *Using APIs to control drones or robots 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*

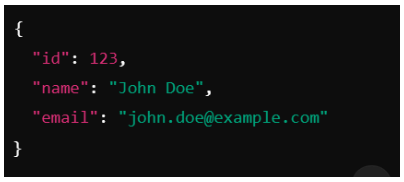
4. API Protocols

Sunday, January 19, 2025 8:31 AM

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
    - Common HTTP methods: GET, POST, PUT, DELETE
    - Use Cases: Web applications, mobile applications, and public APIs (e.g., Twitter API, GitHub API)

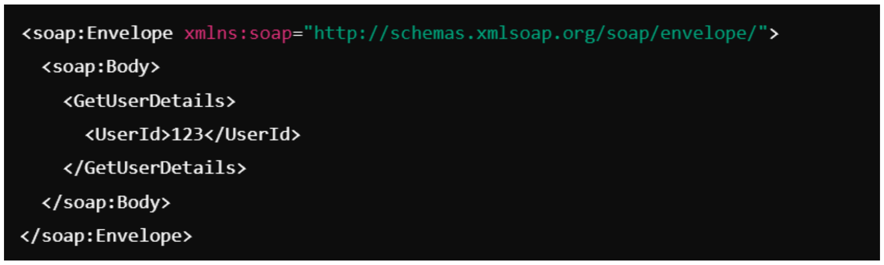


```
GET /users/123 HTTP/1.1
Host: example.com
```



```
{
  "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
   - 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

Feature	REST	SOAP	GraphQL
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

5. Working

Sunday, January 19, 2025 9:50 AM

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

2. **API Endpoint:**
   - The request is sent to a specific URL known as an API endpoint
   - 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

6. API Components

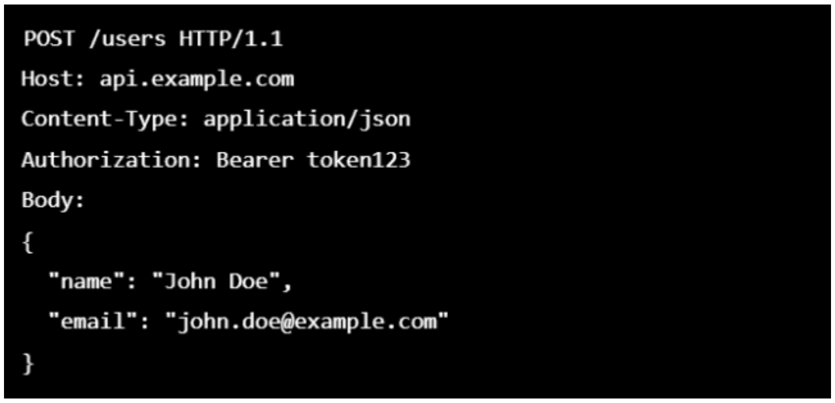
Sunday, January 19, 2025 10:08 AM

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

2. **Endpoint:**

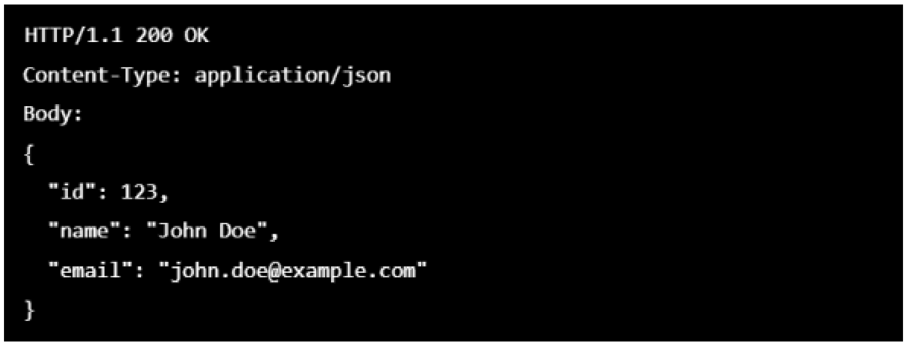
- Message sent by a client to the API to ask for information or perform an operation
- Key components of a request:
  - Method: Specifies the type of operation (e.g., GET, POST, PUT, DELETE)
  - Headers: Provide metadata about the request, such as content type
  - Body: Contains the data being sent to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



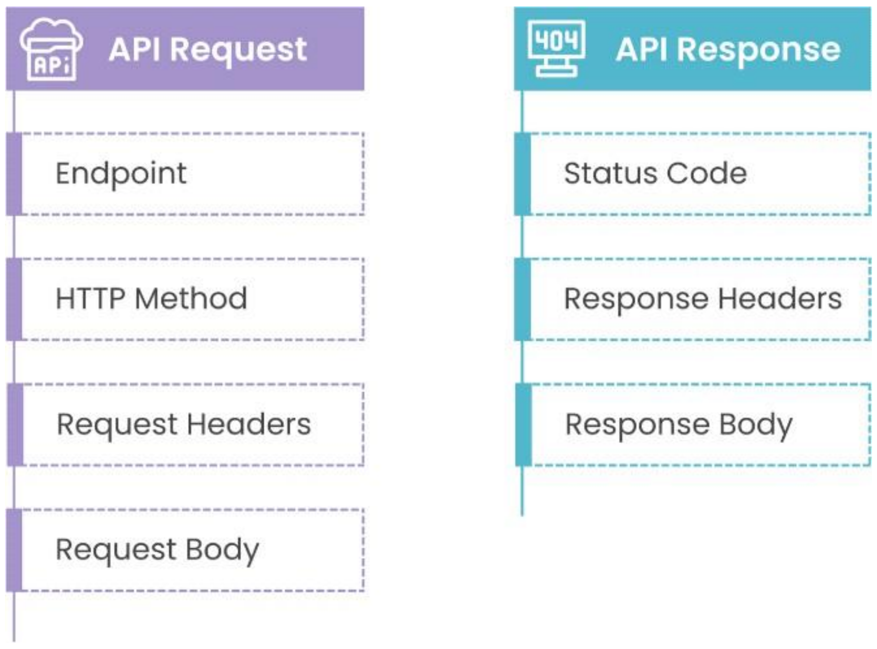
```
POST /users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer token123
Body:
{
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

3. **Response:**

- 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



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



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

7. API Lifecycle

Sunday, January 19, 2025 10:24 AM

- 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

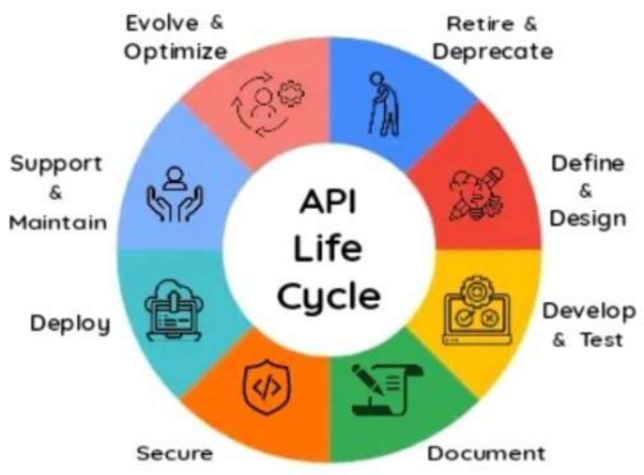
2. **Development:**
    - The development phase involves implementing the API's backend logic and infrastructure
    - Write the server-side code using programming frameworks like Flask, FastAPI, or Express.js
    - Integrate authentication and security mechanisms
    - Perform unit testing to validate 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
    - 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



8. Authentication & Authorization

Sunday, January 19, 2025 11:01 AM

- 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

0. Topics

Thursday, May 8, 2025 11:03 PM

1. **What is FastAPI?**

2. **Key Features**

3. **Architecture**

4. **Installation**

5. **First FastAPI App**

6. **Comparison**

1. What is FastAPI?

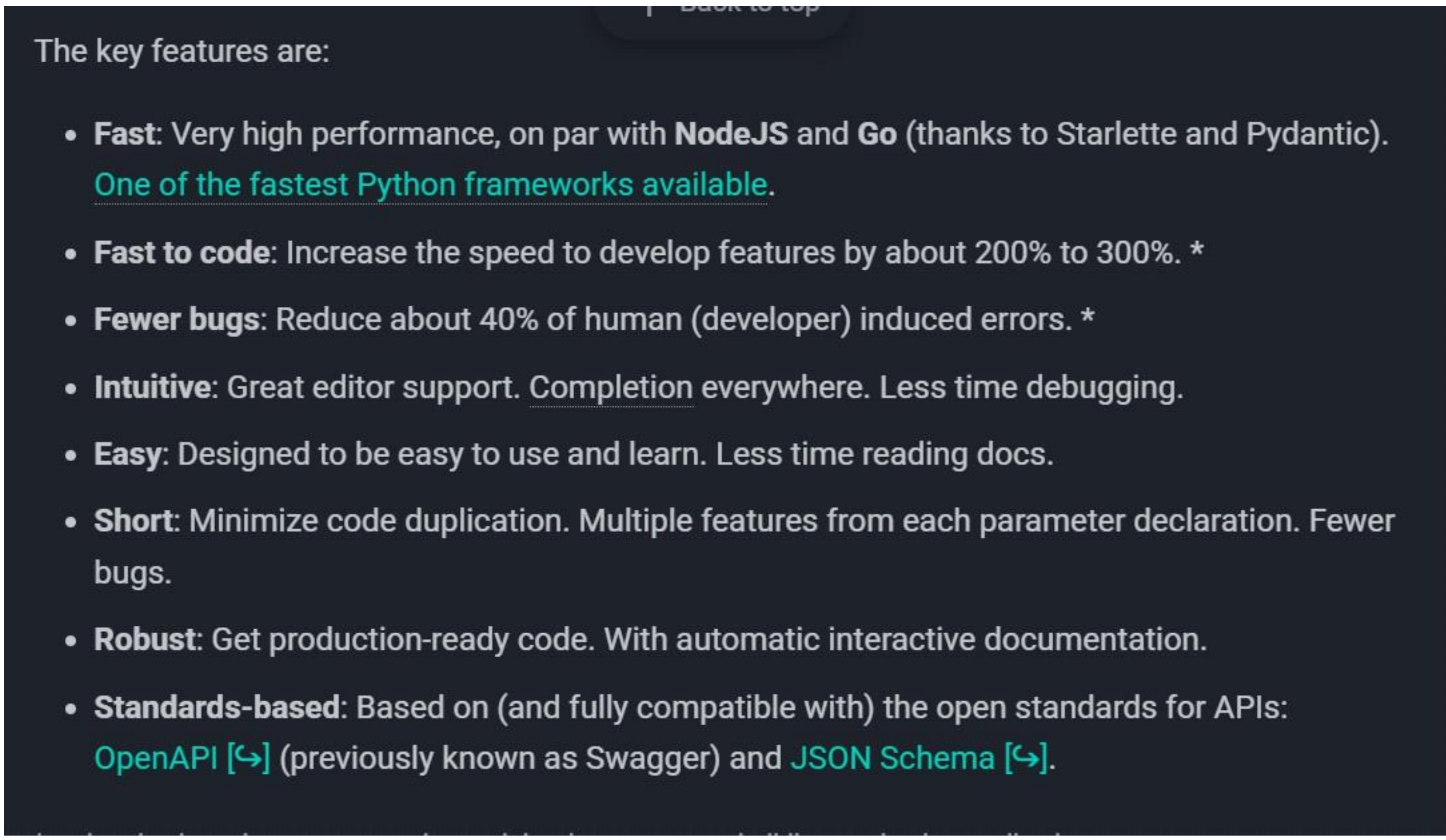
Thursday, May 8, 2025 11:08 PM

- 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+

2. Key Features

Thursday, May 8, 2025 11:13 PM

From the original website:



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

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

3. Architecture

Thursday, May 8, 2025 11:28 PM

FastAPI is Built on Three Core Libraries:

- **Starlette:**
    - 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

4. Installation

Friday, May 9, 2025 2:06 AM

1. **Create/Activate a virtual environment:**

2. **Install FastAPI and Uvicorn:**
    - pip install fastapi uvicorn

5. First FastAPI App

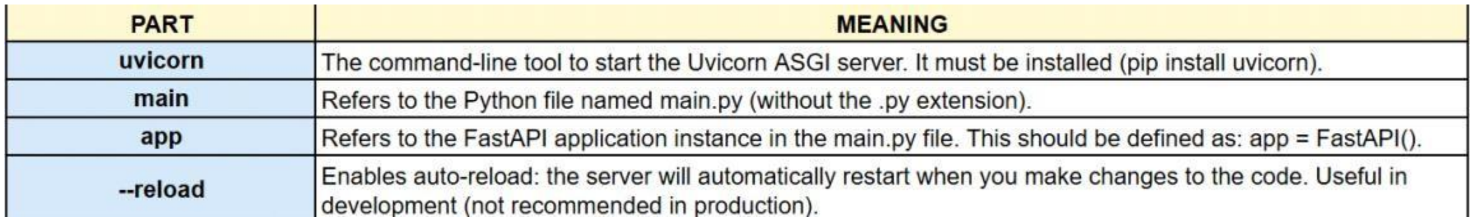
Friday, May 9, 2025 2:09 AM

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

6. Comparison

Friday, May 9, 2025 2:24 AM

FEATURE	FastAPI	Flask	Django	Falcon
Release Year	2018	2010	2005	2013
Asynchronous Support	Native (async/await)	Limited (via extensions)	Partial (since Django 3.1)	Full
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)

0. Topics

Thursday, May 8, 2025 11:03 PM

1. **Creating APIs using FastAPI**
2. **CRUD Operations**
3. **Handling Validations and Errors**
4. **Asynchronous Programming**

1. Creating APIs using FastAPI

Saturday, May 10, 2025 12:50 PM

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
)
```

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()

2. CRUD Operations

Saturday, May 10, 2025 1:04 PM

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

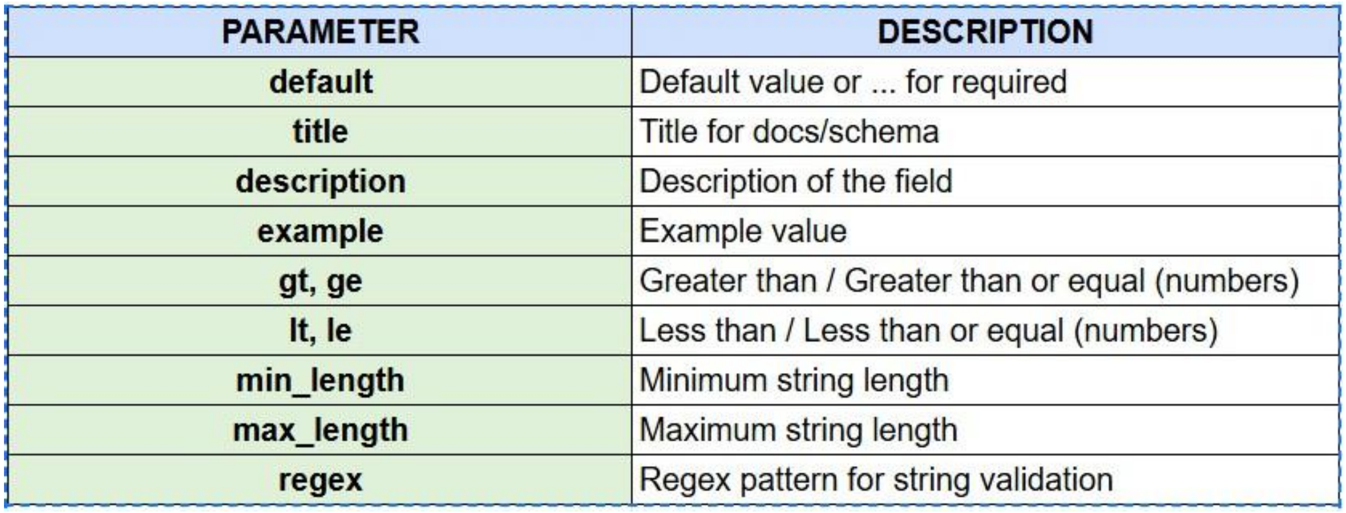
3. Handling Validations and Errors

Monday, May 12, 2025 8:24 AM

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

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

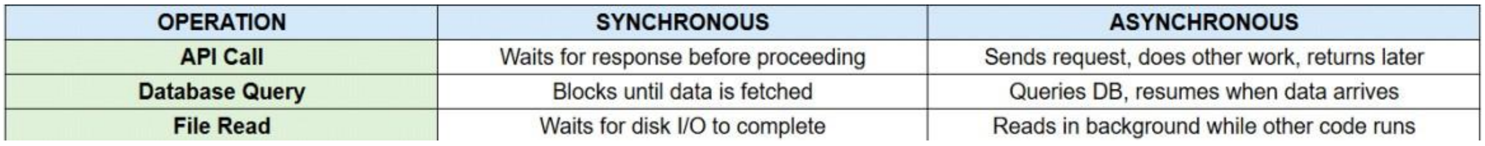
4. Asynchronous Programming

Monday, May 12, 2025 9:16 AM

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

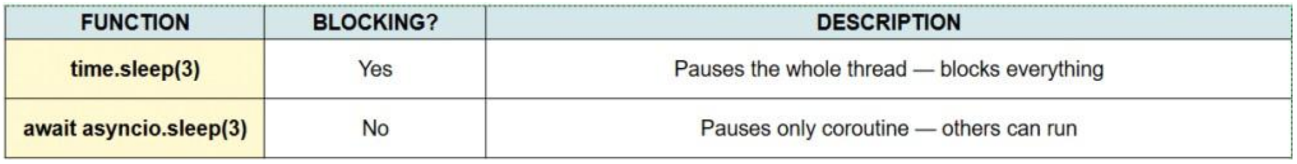
async and await:

In Python, asynchronous code is written using asynciomodule:

- 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**)

0. Topics

Tuesday, May 13, 2025 11:16 PM

1. **Database Basics**
2. **SQLAlchemy Basics**
3. **CRUD app using FastAPI & SQLAlchemy**

1. Database Basics

Tuesday, May 13, 2025 11:22 PM

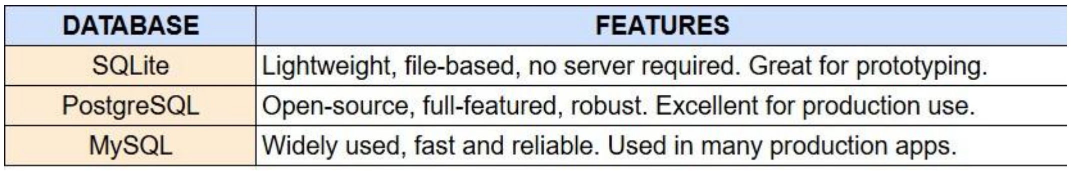
What is a Database?

- 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
  - 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

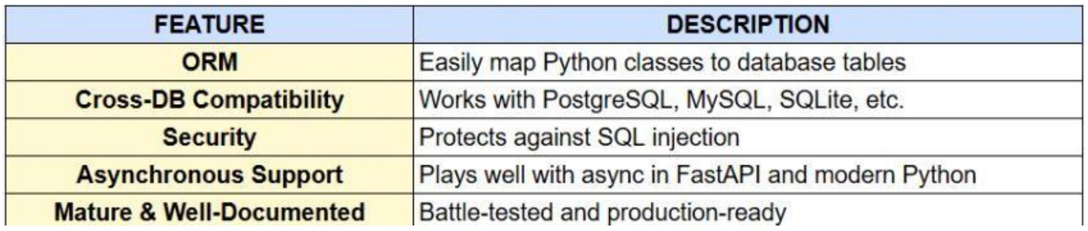
2. SQLAlchemy Basics

Tuesday, May 13, 2025 11:39 PM

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]

3. CRUD App

Wednesday, May 14, 2025 5:24 AM

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

3.1 - database.py

Saturday, May 17, 2025 12:48 PM

- Helps setup database connection and provide foundational components for ORM
- Centralizes DB setup, making it reusable across the app for sessions and models

create_engine():

  - Establishes the connection to the database
  - Here, it connects to a SQLite file named test.db

connect_args:

  - SQLite-specific to allow connection sharing across threads

sessionmaker:

  - Helps create new database sessions
  - Each session represents a transactional scope to the DB
  - **autoflush=False**:
    - SQLAlchemy will not automatically flush changes to the DB unless explicitly committed or refreshed
  - **autocommit=False**:
    - Disables automatic commit after each query
    - Commit manually to control transactions

declarative_base():

  - Creates a base class for models to inherit from, linking the Python classes with DB tables

3.2 - models.py

Saturday, May 17, 2025 1:02 PM

Defines the **Employee** model, mapping the Python class to the **employees** table in the DB

- Helps auto-generate fields like **ID**
- Can make different attributes optional or use default values for specific tasks as required
- Helps add validations specific to one operation without affecting others
- Having explicit classes for each action makes the code self-explanatory
- Makes it easier to debug and extend

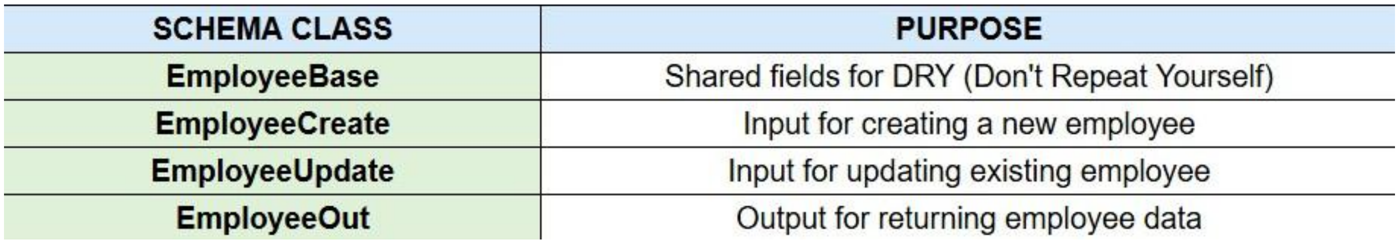
3.3 - schemas.py

Saturday, May 17, 2025 1:04 PM

- Defines Pydantic models (schemas) for request validation and response formatting
- Promotes data consistency and validation against invalid inputs

orm_mode = True:

- ○ Allows Pydantic to read data directly from ORM objects (SQLAlchemy models)
- ○ Enables smooth conversion to JSON



SCHEMA CLASS	PURPOSE
EmployeeBase	Shared fields for DRY (Don't Repeat Yourself)
EmployeeCreate	Input for creating a new employee
EmployeeUpdate	Input for updating existing employee
EmployeeOut	Output for returning employee data

3.4 - crud.py

Saturday, May 17, 2025 1:07 PM

- This module encapsulates all database operations for **Employee**
- Abstracts away raw DB queries from the API routes, keeping the code modular
- Keeps DB logic separate and reusable, making the app cleaner and easier to maintain

refresh():

  - reloads the object from DB to get autogenerated fields (**id**)

3.5 - main.py

Saturday, May 17, 2025 1:07 PM

- This is where the API is defined, exposing endpoints for the end users
- Denotes the entry point of the API server
- Defines the FastAPI app and all associated routes

create_all(bind=engine):

- creates DB tables based on your models if they don't exist

get_db():

- Dependency to provide a DB session to routes
- Uses the **yield** keyword to make it a generator
- Helps FastAPI manage the opening and closing of DB connections cleanly

0. Topics

Sunday, May 25, 2025 11:14 AM

1. **Model Serialization**
2. **Pickle vs Joblib**
3. **Designing Model I/O Schemas**
4. **Serving ML Models with FastAPI**
5. **Handling Batch Predictions**

1. Model Serialization

Sunday, May 25, 2025 11:23 AM

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
- Keras (**.h5, .keras**)
- 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:

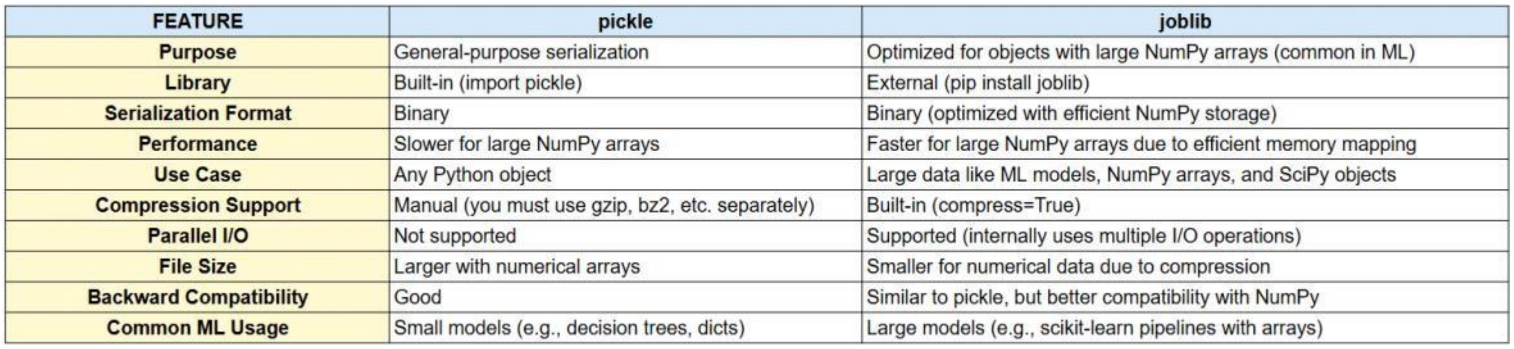
- Serialization enables seamless deployment of ML models into real-world systems like REST APIs, web applications, dashboards, or edge devices
- Allows decoupling of training and inference phases:
    - Training happens offline (e.g., Jupyter notebook)
    - Inference happens online (e.g., real-time 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

2. Pickle vs Joblib

Sunday, May 25, 2025 12:38 PM



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)

3. Designing Model I/O Schemas

Sunday, May 25, 2025 12:24 PM

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

4. Serving ML Models

Sunday, May 25, 2025 12:39 PM

Why Serve ML Models via FastAPI?

1. **Decoupling Model from Application Logic:**
    - Keeps ML logic separate from UI or business logic
    - Enables modular, reusable models that can be consumed by multiple applications (web apps, mobile apps, internal tools, etc.) via HTTP requests

2. **Real-Time Predictions:**
    - Clients can send input and receive predictions instantly via a /predict endpoint
    - Essential for use-cases like **fraud detection**, **recommendation engines**, or **medical triaging** where immediate inference is needed

3. **Platform-Agnostic Integration:**
    - The model is accessible through a standard REST API
    - Any frontend, mobile app, or backend service, regardless of language (JS, Java, etc.), can access the model using HTTP

4. **Production-Readiness:**
    - FastAPI supports ASGI, async I/O, and is built for performance
    - FastAPI is faster than Flask, with excellent concurrency, making it scalable for real-world traffic

5. **Built-in Validation with Pydantic:**
    - Ensures clean, validated data before it reaches the model (**InputSchema**)
    - The response is returned in a structured format (**OutputSchema**)
    - Prevents runtime errors due to bad input, reduces bugs, and improves model reliability

6. **Docker & Cloud Friendly:**
    - FastAPI apps can be containerized and deployed on AWS, Azure, GCP, Hugging Face, Render, etc.
    - Perfect fit for **CI/CD pipelines**, **Kubernetes**, and serverless deployment models

7. Scalable Infrastructure:

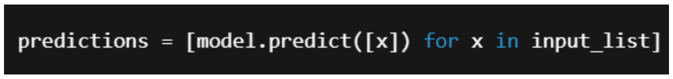
- Works with ASGI servers like Uvicorn for high-concurrency
- Allows to serve thousands of predictions per second with proper load balancing

5. Handling Batch Predictions

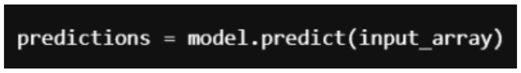
Sunday, May 25, 2025 2:03 PM

Vectorized Predictions for Speed:

- When making predictions using a machine learning model, especially for multiple data points, it's inefficient to loop over each input and call the **.predict()** method one-by-one
- Instead, we should leverage **vectorized predictions**, i.e., send a whole batch into the model at once
- This drastically improves performance due to:
    - Optimized linear algebra libraries under the hood
    - Reduced I/O overhead
    - Parallelized CPU/GPU execution



```python
predictions = [model.predict([x]) for x in input_list]
```

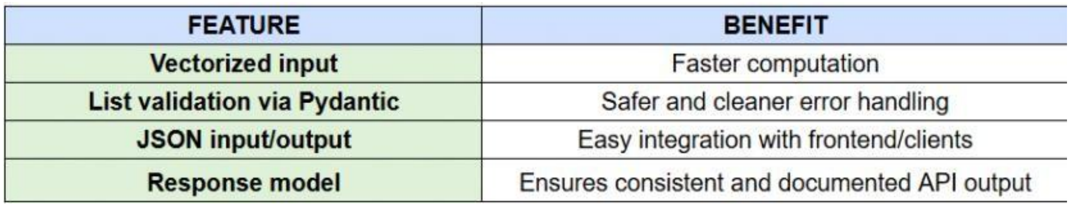


```python
predictions = model.predict(input_array)
```

Accepting List of Inputs:

- To handle batch predictions via API, the endpoint should accept a list of input objects (typically JSON dictionaries)
- This means instead of a single input, the user will **POST** a list of inputs to the endpoint

Benefits:



FEATURE	BENEFIT
Vectorized input	Faster computation
List validation via Pydantic	Safer and cleaner error handling
JSON input/output	Easy integration with frontend/clients
Response model	Ensures consistent and documented API output

0. Topics

Thursday, May 29, 2025 10:03 PM

1. **Middlewares**

2. **Dependency Injection**

3. **JWT Authentication**

4. **Managing API Keys**

5. **Best Practices**

1. Middleware

Thursday, May 29, 2025 10:08 PM

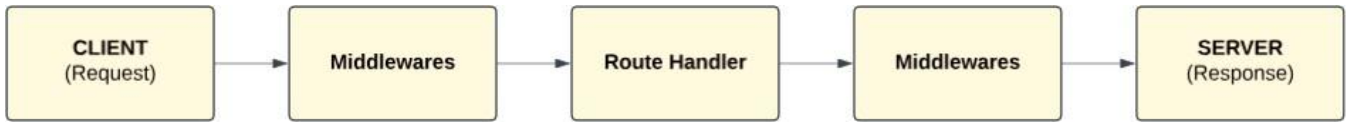
What is a Middleware?

- 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**

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:



1.1 - Built-in Middlewares

Thursday, May 29, 2025 11:16 PM

1. **CORS Middleware:**
    - Cross-Origin Resource Sharing
    - Allows cross-origin requests from browsers
    - Injects the proper headers in the HTTP responses so that compliant browsers will permit—or reject—those cross-origin calls
    - Helps declare exactly which external origins, HTTP methods, headers (and even whether credentials/cookies) are allowed
        - **allow_origins**: which domains are permitted to make browser-based requests
        - **allow_credentials**: whether to let the browser send cookies
        - **allow_methods**: which actions will be accepted
        - **allow_headers**: which headers will be accepted

2. **GZip Middleware:**
    - Used to compress HTTP responses before sending them to the client
    - Reduces the size of data transferred over the network
    - Faster page loads
    - Lower bandwidth usage
    - Improved performance, especially over slower networks

3. **HTTPSRedirectMiddleware:**
    - Ensures that all incoming HTTP requests are automatically redirected to HTTPS
    - Enforcing HTTPS is a modern security standard, which is preferred by many platforms (browsers, Google SEO ranking)

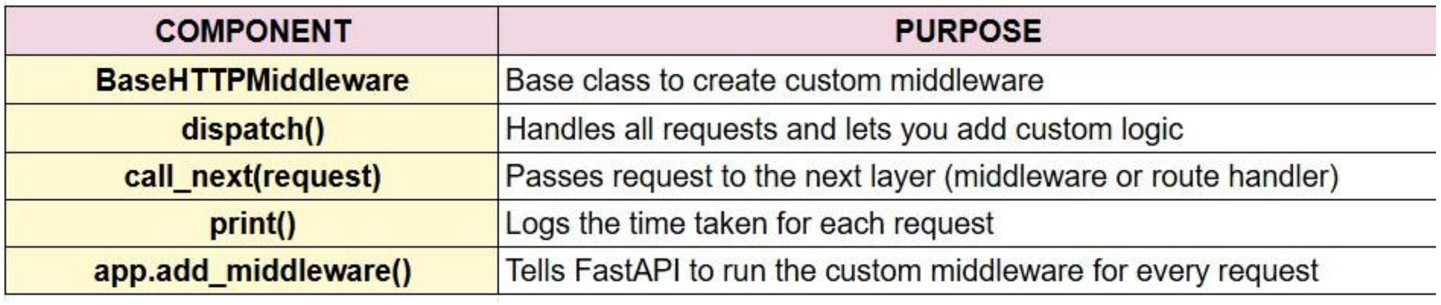
1.2 - Custom Middlewares

Thursday, May 29, 2025 11:17 PM

Implement a Custom Middleware:

- Define a class for the middleware
- Implement the dispatch method which contains the logic
- Register the middleware
- Add a route to handle requests
- Run the application
- Observe the terminal logs

Summary:



COMPONENT	PURPOSE
BaseHTTPMiddleware	Base class to create custom middleware
dispatch()	Handles all requests and lets you add custom logic
call_next(request)	Passes request to the next layer (middleware or route handler)
print()	Logs the time taken for each request
app.add_middleware()	Tells FastAPI to run the custom middleware for every request

2. Dependency Injection

Friday, May 30, 2025 12:06 AM

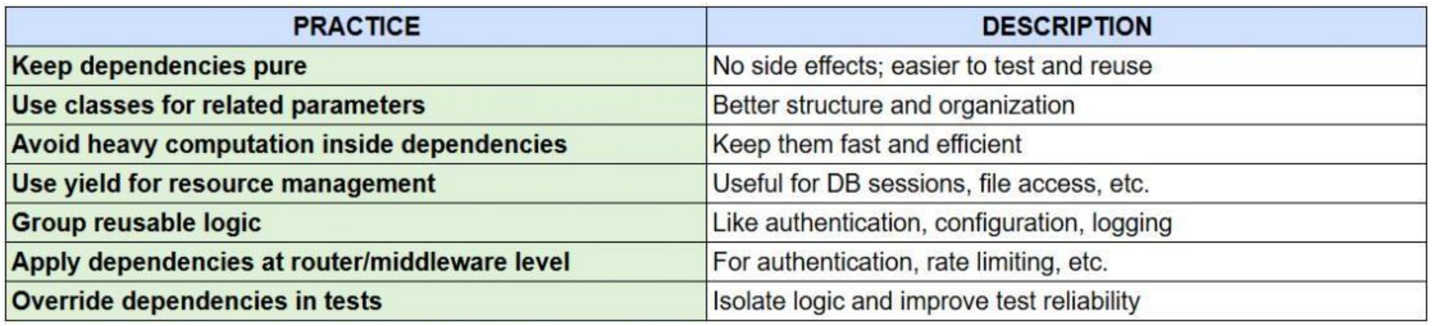
What is Dependency Injection?

- 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

2.1 - Database Connections

Friday, May 30, 2025 12:37 AM

- Database connection is required and must be established before any database operation is performed
- **get_db()** is a dependency function
- **Depends(get_db)** tells FastAPI to call **get_db()** before the request and inject the result into the **db** parameter

2.2 - Configuration Management

Friday, May 30, 2025 12:37 AM

- Provides a centralized way to store and access configuration values (ex. **api_key**, **debug**)
- Injects these values into FastAPI route handlers using **Depends()**
- All config values are encapsulated in one place (**Settings**), improving maintainability
- API keys, database URLs, etc., can be loaded here securely

When the /config/ endpoint is called:

- FastAPI calls **get_settings()**
- The result is passed into the **get_config** function as the **settings** argument
- The function returns a dictionary containing the **api_key**

2.3 - User Authentication

Friday, May 30, 2025 12:37 AM

- The main idea is to protect an API route so that only authenticated users with a valid token can access it

- **oauth2_scheme** is a dependency provided by FastAPI:
  - Extracts the token from **Authorization** header
  - Automatically makes the token string available for further use

- **decode_token(token)** is a placeholder to implement logic:
  - Validate the token
  - Decode the token
  - Return the user data embedded in the token

- **get_current_user()**: Makes user info available to routes

- **Depends(get_current_user)**: Protects the route—only accessible if authenticated

- If no token is provided or it's invalid, FastAPI automatically raises a 401 error

- If the token is valid, the endpoint returns the username of the logged-in user

3. JWT Authentication

Friday, May 30, 2025 10:04 PM

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**

3.1 - auth.py

Friday, May 30, 2025 10:21 PM

Purpose:

- This module handles **JWT** creation and verification
- This is the security engine of the application
- This is essential for:
    - Creating **access tokens** after successful user authentication
    - Verifying and decoding **access tokens** to authorize users accessing **protected routes**

Constants:

- **SECRET_KEY:**
    - The application's secret key
    - Must be kept confidential (e.g., in a .env file)
    - Used to sign tokens
- **ALGORITHM:**
    - Specifies the signing algorithm
    - **HS256** is widely used (HMAC with SHA-256)
- **ACCESS_TOKEN_EXPIRE_MINUTES:**
    - Token expiry time (in minutes)
    - Controls how long a token remains valid

Functions:

- **create_access_token:**
    - Generates a JWT access token using user data (typically the username or user ID)

    - Makes a copy to avoid mutating the original
    - Adds an expiry claim to the token payload; JWTs must contain expiration for security
    - Encodes and signs the token using the secret and algorithm

- **verify_token:**
    - Verifies and decodes a token to extract user identity and check validity
    - Decodes the token using the secret key and expected algorithm
    - Raises **JoseError** if signature or expiry is invalid
    - Returns the user identifier extracted from token

3.2 - models.py

Friday, May 30, 2025 10:21 PM

Why use UserInDB(User)?

- Reuse **username** and **password** fields
- Add **hashed_password** for DB logic only
- Clearly separate:
  - What the client sends (**User**)
  - What the server works with internally (**UserInDB**)
- This structure improves code reusability, security, and clarity

3.3 - utils.py

Friday, May 30, 2025 10:21 PM

Purpose:

- The purpose of this module is to encapsulate utility functions related to user data and password hashing/verification
- Acts as a helper module that makes the codebase more modular and reusable

3.4 - main.py

Friday, May 30, 2025 10:21 PM

Purpose:

- Connects all the pieces (auth.py, utils.py, models.py) and defines the API endpoints
- The login endpoint issues JWT tokens
- A protected route is created, accessible only with a valid token

Endpoints:

- **/token:**
  - This endpoint authenticates the user
  - Try to fetch the user using **get_user()**
  - If not found, raise an error with status code 400
  - Verify the password using **verify_password()**
  - If incorrect, raise an error with status code 400
- **/users:**
  - Extracts the token from the **Authorization** header and injects it into the **token** parameter
  - Decodes and validates the JWT
  - Returns the username encoded in the token

Execution:

- Open the Swagger UI
- Execute the endpoint **/token** and give the below details:
  - **Username**: johndoe
  - **Password**: secret123
- Save the returned **access_token**
- Click the **Authorize** button (top-right of Swagger UI) and paste the token

- Execute the endpoint **/users** to receive the username

3.5 - Workflow

Friday, May 30, 2025 11:40 PM

1. **User Login and Token Issuing (/token):**
   - The client sends a **POST** request to with **username** and **password**
   - The app checks if the user exists (**get_user** from a fake DB)
   - It verifies the plain password against the stored hashed password (**verify_password**)
   - If valid, the app creates a JWT access token (using a secret key and algorithm)
   - It sends back the token in the response, which the client will use for subsequent requests
   - This step shows **credential verification and token issuance** — the core of logging in

2. **Token Usage for Protected Routes (/users):**
   - This endpoint requires a token — it uses **Depends(oauth2_scheme)** and extracts the token from the **Authorization** header
   - The token is verified (**verify_token**) and decodes the JWT to confirm it's valid and not expired
   - The username is extracted from the token payload
   - The endpoint returns info about the current user (in this case, the **username**)
   - This step shows **token validation and access control** — only users with a valid token can access this protected endpoint

3. **Password Hashing & Verification:**
   - User passwords are never stored or transmitted in plain text
   - Passwords are hashed (using **bcrypt** via **passlib**) when stored
   - On login, the plain password entered is verified against the stored hash using **verify_password**
   - This ensures **passwords are kept secure**

4. Managing API Keys

Friday, May 30, 2025 11:53 PM

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

Implementing API Keys with FastAPI via:

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

4.1 - Headers

Saturday, May 31, 2025 6:29 AM

- **Implement the application**

- **Run the server**

- **Execute the curl command:**
    - **curl -H "api-key: my-secret-key" http://localhost:8000/get-data**

4.2 - Environment Variables

Saturday, May 31, 2025 6:29 AM

- **Create a .env file and include the value:**
  - **api_key=my-secret-key**

- **Implement the application**
  - Install pydantic-settings (if not already)
  - **pip install pydantic-settings**

  - **BaseSettings** is a great utility that reads environment variables and casts them to the correct type
  - The **Config** class tells Pydantic to load from a **.env** file
  - **settings.api_key** can now be accessed securely in the app

- **Run the server**

- **Execute the curl command:**
  - **curl -H "api-key: my-secret-key" http://localhost:8000/get-data**

5. Best Practices

Friday, May 30, 2025 11:54 PM

- Use **HTTPS** in production
- 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

0. Topics

Monday, June 9, 2025 9:46 PM

1. Importance of Testing APIs

2. Types of Tests

3. Mock ML Models

4. Common API Errors

5. Debugging Techniques

1. Importance of Testing APIs

Monday, June 9, 2025 9:53 PM

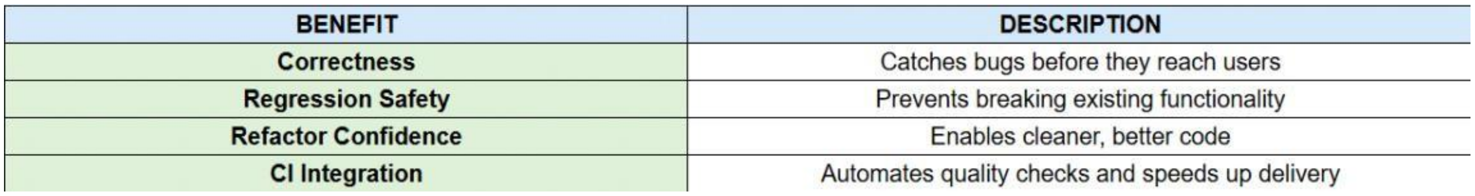
- Testing is a non-negotiable aspect of modern API development
- It ensures that the web application functions correctly, is reliable, and behaves consistently even as you scale the applicat ion

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:**
    - 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:**
    - 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

2. Types of Tests

Monday, June 9, 2025 10:08 PM

- Testing at different layers is essential for ensuring both small units and the system as a whole work correctly
- Each type of test has a distinct purpose, scope, and cost to run

1. Unit Tests:

- Focuses on a single unit of logic—typically a **function** or **method**
- Should not depend on external services like a database, file system, or network
- Fast to run and ideal for **TDD (Test-Driven Development)**
- **When to use:**
  - Testing utility functions
  - Business logic (e.g., tax calculation, discount rules)
  - Schema validation (e.g., Pydantic model validators)
- **How pytest works:**
  - Use the command **pytest tests/**
  - **Pytest** scans the **tests/** folder for any python files that start with **test_** or end with **_test.py**
  - Inside those files, it looks for **functions** that start with **test_**
  - After all tests are run, pytest summarizes the results

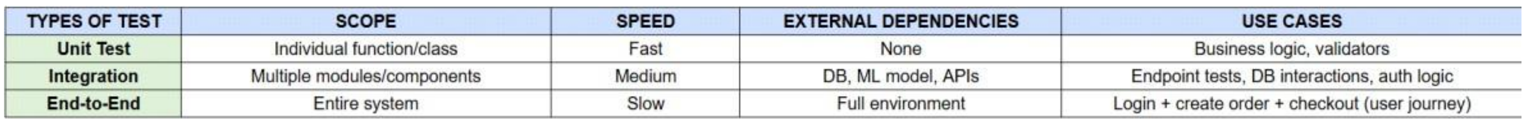
2. Integration Tests:

- Involves multiple units working together: **database + app**, **API + ML model**, etc.
- Typically uses in-memory or test database environments
- More realistic than **Unit Tests**, but also slower
- **When to use:**
  - Testing API endpoints
  - Testing FastAPI dependencies (like database or ML models)
  - Validating middleware behavior
  - Ensuring JWT authentication works across endpoints

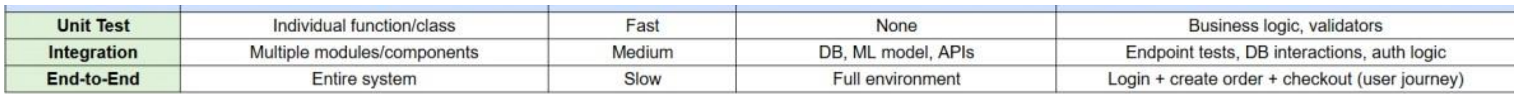
3. End-to-End (E2E) Tests:

- Simulates real user flows like login, submitting a form, or making a prediction
- Runs the system as a black box (i.e., without knowing internal code)
- May interact with UI, frontend, or API
- **When to use:**
  - Pre-deployment checks
  - Validating critical user workflows like sign-up/login/purchase
  - Testing deployment setup and network configurations

Summary:



TYPES OF TEST	SCOPE	SPEED	EXTERNAL DEPENDENCIES	USE CASES
Unit Test	Individual function/class	Fast	None	Business logic, validators
Integration	Multiple modules/components	Medium	DB, ML model, APIs	Endpoint tests, DB interactions, auth logic
End-to-End	Entire system	Slow	Full environment	Login + create order + checkout (user journey)



Unit Test	Individual function/class	Fast	None	Business logic, validators
Integration	Multiple modules/components	Medium	DB, ML model, APIs	Endpoint tests, DB interactions, auth logic
End-to-End	Entire system	Slow	Full environment	Login + create order + checkout (user journey)

3. Mock ML Models

Monday, June 9, 2025 11:14 PM

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

Why Mock ML Models?

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

Summary of Mocking in ML API Testing:

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)

3.1 - Demo

Monday, June 9, 2025 11:37 PM

File Structure:

- **main.py**
- **model.py**
- **test_main.py**
- **training.ipynb**

Control Flow:

- Uses the **patch()** method from **unittest.mock** to mock the real ML model's **.predict()** method
- The mocked method always returns **[99]**
- Sends a **POST** request to the **/predict** endpoint using **TestClient**
- Asserts that the endpoint returns a **200 OK** with a prediction of **99**
- Ensures that the mocked **.predict()** was called with the expected input

4. Common API Errors

Tuesday, June 10, 2025 10:13 PM

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

5. Debugging Techniques

Tuesday, June 10, 2025 10:30 PM

FastAPI makes Debugging easier through:

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

5.1 - Logging

Tuesday, June 10, 2025 10:35 PM

- Logging is a fundamental debugging practice in development
- Instead of using **print()**, we use Python's **logging** module, which is more powerful and configurable
- Logging helps with:
    - Track what part of the app was accessed
    - Log variables and errors
    - Retain logs in files or forward them to monitoring systems

Log Levels:

- **DEBUG:** Low-level system information
- **INFO:** Routine information like successful requests
- **WARNING:** Something unexpected happened
- **ERROR:** A serious problem
- **CRITICAL:** Severe errors that cause premature termination

5.2 - Exception Handling

Tuesday, June 10, 2025 10:35 PM

FastAPI lets developers define custom exception handlers to:

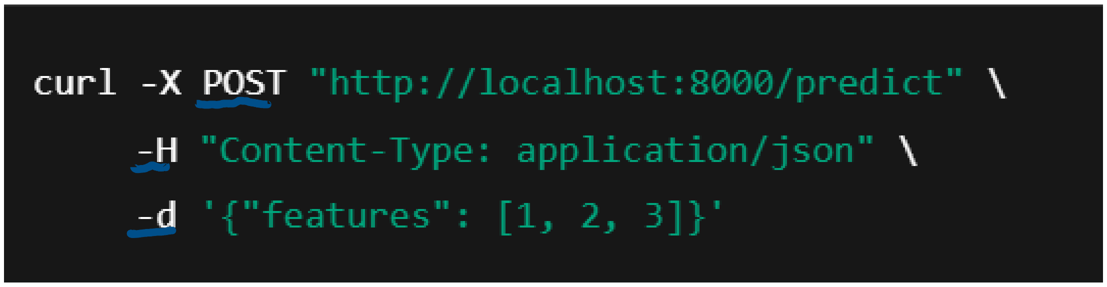
- **Catch unhandled errors**
- **Return consistent and informative error messages**
- **Log errors for debugging and auditing**

5.3 - curl

Tuesday, June 10, 2025 10:35 PM

- Send Requests with headers, payload, methods (GET, POST, PUT, DELETE)
- Inspect responses (status codes, headers, content)
- Helpful in isolating whether issues are in the **backend or client**

Syntax:



```
curl -X POST "http://localhost:8000/predict" \
    -H "Content-Type: application/json" \
    -d '{"features": [1, 2, 3]}'
```

- **-X POST:** Specifies the HTTP method
- **-H:** Adds headers
- **-d:** Sends request body data

5.4 - Configurations

Tuesday, June 10, 2025 10:35 PM

Running **Uvicorn** with **--reload** and **--debug** enables live reloading and better error visibility during development:

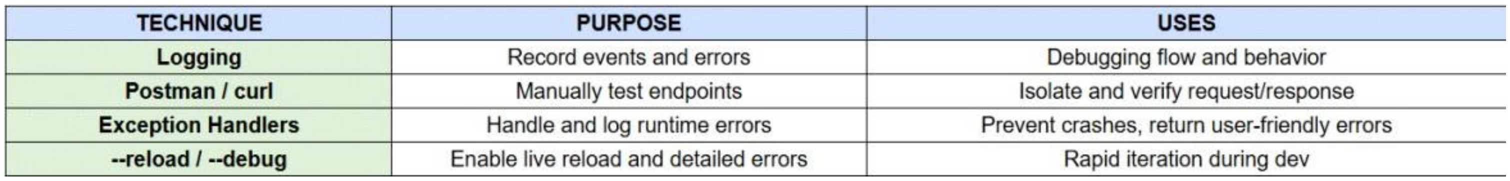
- **--reload:** Automatically restarts the server when you change code (development only)
- **--debug:** Enables verbose output and stack traces in logs (not for production)

Syntax:

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

5.5 - Summary

Tuesday, June 10, 2025 11:16 PM



TECHNIQUE	PURPOSE	USES
Logging	Record events and errors	Debugging flow and behavior
Postman / curl	Manually test endpoints	Isolate and verify request/response
Exception Handlers	Handle and log runtime errors	Prevent crashes, return user-friendly errors
--reload / --debug	Enable live reload and detailed errors	Rapid iteration during dev

0. Topics

Monday, June 16, 2025 10:16 PM

1. **Caching**
2. **Caching with Redis**
3. **Redis with FastAPI**
4. **Profiling FastAPI apps**
5. **Benchmarking APIs**
6. **Monitoring APIs**

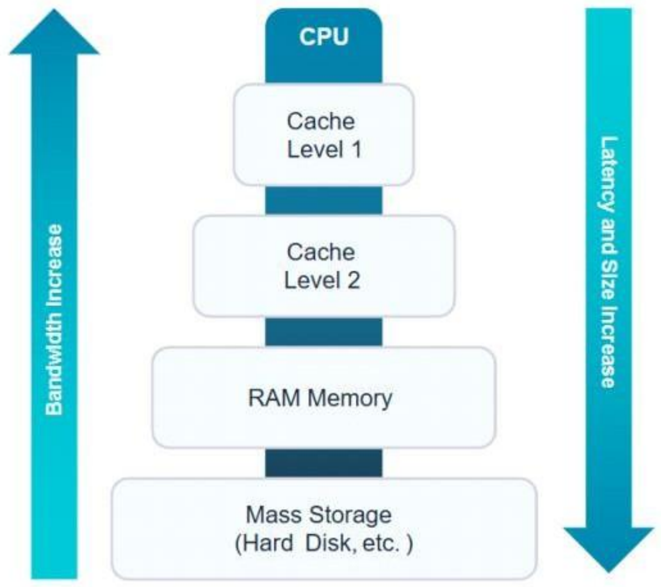
1. Caching

Monday, June 16, 2025 10:20 PM

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



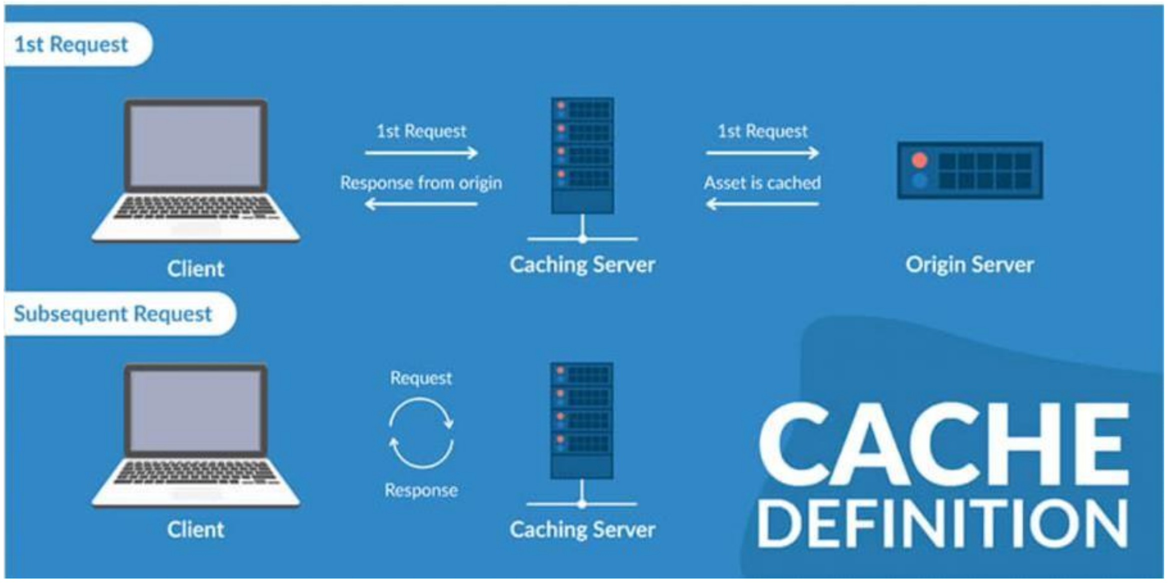
An Overview of Memory Caching

1.1 - Importance of Caching

Monday, June 16, 2025 10:31 PM

Why is Caching Important?

- **Reduces Latency:** Cached responses are served from nearby or in-memory storage, which is significantly faster than from a database or an external API call
- **Improves Performance:** Applications become more responsive since frequently requested data is readily available
- **Reduces Load on Backend:** By reducing repeated data fetches or computations, the pressure on databases, ML models, or third-party APIs is minimized
- **Scalability:** Helps applications scale better under high load, as the same data doesn't need to be processed repeatedly

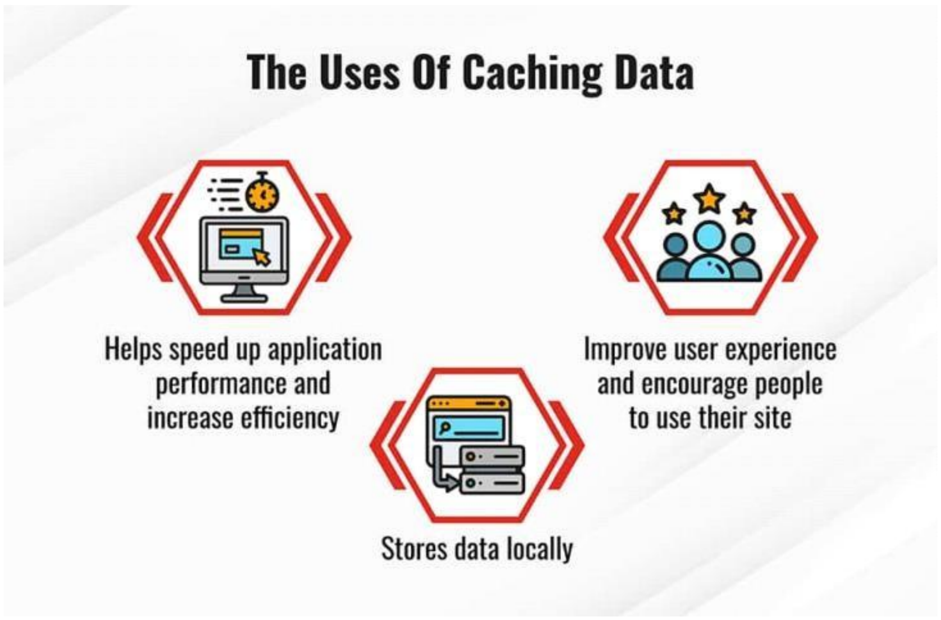


1.2 - Use Cases

Monday, June 16, 2025 10:32 PM

Common Use Cases of Caching:

- **Web content:** Static assets (images, CSS, JS files) are cached by browsers or **CDNs** (Content Delivery Network)
- **Databases:** Frequently queried data are cached to avoid hitting the DB each time
- **API responses:** Slow or rate-limited third-party API calls are cached to avoid repeated calls
- **ML Predictions:** Expensive predictions (e.g., fraud scores, recommendations) are cached for identical inputs
- **Session Data:** In web apps, user sessions are often stored in a cache like **Redis** for fast retrieval


The Uses Of Caching Data
Helps speed up application performance and increase efficiency
Improve user experience and encourage people to use their site
Stores data locally

1.3 - Types of Caching

Monday, June 16, 2025 10:32 PM

Types of Caching:

- **Client-side caching:** Done in browsers or frontends using mechanisms like HTTP headers (**Cache-Control**)
- **Server-side caching:** Caching happens on the server using tools like **Redis**, **Memcached**, or **in-memory dictionaries**
- **CDN caching:** Content Delivery Networks cache static resources close to users for faster load times
    - A **Content Delivery Network (CDN)** is a network of geographically distributed servers that work together to deliver digital content (websites, videos, images, scripts, etc.) to users more efficiently and reliably
    - Instead of every user fetching content from a single central server (which can be slow or get overloaded), a CDN caches and serves content from a server that is geographically closer to the user
    - This reduces latency, speeds up loading times, and improves scalability and availability

1.4 - Key Considerations

Monday, June 16, 2025 10:32 PM

Key Considerations with Caching:

- **Cache Invalidation:** When data changes, how do you ensure the cache is updated? This is a crucial challenge in caching systems
- **Eviction Policies:** Caches have limited memory, so older or less-used data must be removed using strategies like:
    - **LRU** (Least Recently Used)
    - **LFU** (Least Frequently Used)
    - **FIFO** (First In First Out)
- **Consistency:** Make sure cached data doesn't become stale or inconsistent with the source

1.5 - Tools

Monday, June 16, 2025 10:33 PM

Commonly Used Tools:

- **Redis:** An in-memory key-value store, widely used for caching due to its speed and flexibility
- **Memcached:** Lightweight and fast in-memory caching system
- **CDNs:** Used to cache and serve static content globally (e.g., **Cloudflare**, **Akamai**)
- **Local Memory Cache:** Python dictionaries or FastAPI **lru_cache** decorators for lightweight scenarios

2. Caching with Redis

Monday, June 16, 2025 11:05 PM

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

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:

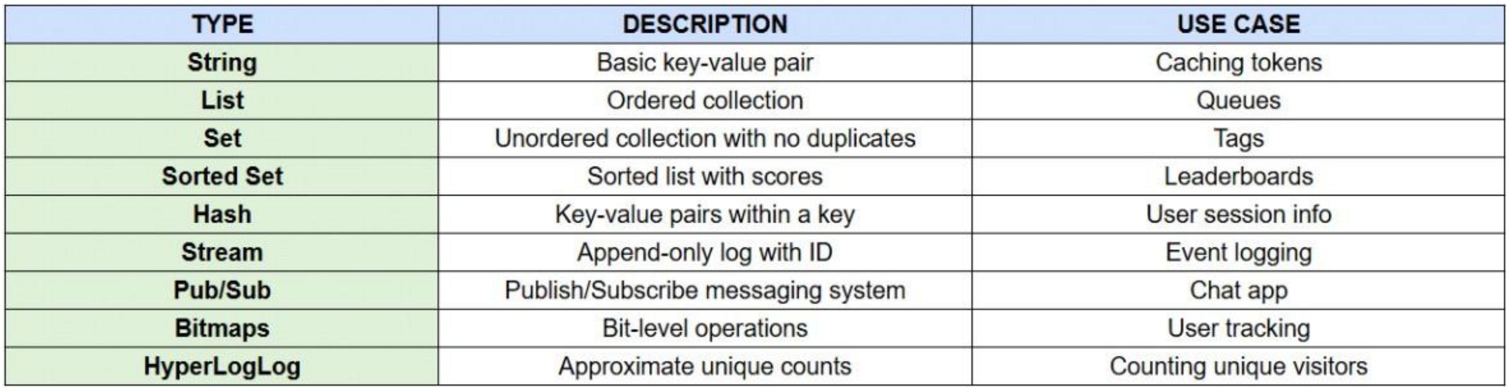
Though Redis is an in-memory store, it supports data persistence through:

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

2.1 - Redis Data Structures

Monday, June 16, 2025 11:11 PM

Redis supports a wide variety of data structures:



TYPE	DESCRIPTION	USE CASE
String	Basic key-value pair	Caching tokens
List	Ordered collection	Queues
Set	Unordered collection with no duplicates	Tags
Sorted Set	Sorted list with scores	Leaderboards
Hash	Key-value pairs within a key	User session info
Stream	Append-only log with ID	Event logging
Pub/Sub	Publish/Subscribe messaging system	Chat app
Bitmaps	Bit-level operations	User tracking
HyperLogLog	Approximate unique counts	Counting unique visitors

2.2 - Use Cases

Monday, June 16, 2025 11:14 PM

Common use-cases of Redis:

- **Caching** (e.g., ML predictions, DB queries)
- **Session Management** (e.g., storing user sessions in web applications)
- **Rate Limiting** (e.g., API throttling)
- **Real-time Analytics**
- **Leaderboards and Ranking Systems** (sorted sets)
- **Pub/Sub Messaging**

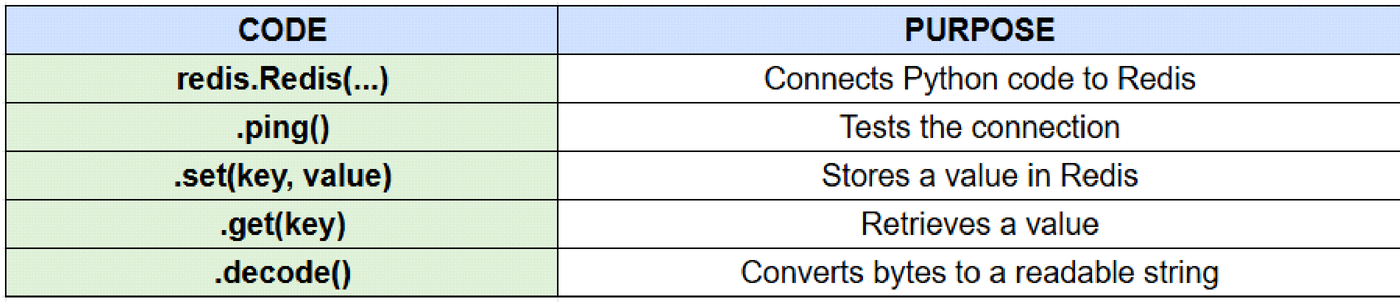
2.3 - Setup

Monday, June 16, 2025 11:25 PM

1. **Install Redis Server (One-time setup):**
    - **Run command in bash: docker run -d -p 6379:6379 redis**

2. **Install Redis Python Client:**
    - **There're generally two options:**
        - **redis-py: pip install redis**
        - **aioredis: pip install redis[async]** (recommended)

3. **Test Redis Setup Using Python:**



CODE	PURPOSE
redis.Redis(...)	Connects Python code to Redis
.ping()	Tests the connection
.set(key, value)	Stores a value in Redis
.get(key)	Retrieves a value
.decode()	Converts bytes to a readable string

3. Redis with FastAPI

Tuesday, June 17, 2025 10:25 PM

1. **Caching ML Predictions**

2. **Caching DB Query Results**

3. **Caching External API Call**

3.1 - Caching ML Predictions

Tuesday, June 17, 2025 10:27 PM

1. **Start Redis with Docker:**
   - **docker run -d -p 6379:6379 redis**
2. **Start the application**
3. **Test with identical inputs**

Code:

- **to_list():**
  - Converts features into a list so they can be passed to the model
- **cache_key():**
  - Serializes the input into JSON (sorted for consistency)
  - Hashes it using **SHA-256** to generate a unique Redis key like **predict:<hash>**

App Workflow:

- Input received at **/predict** as JSON and validated against **IrisFlower**
- A unique cache key is generated using a **SHA-256** hash of the input
- The app checks if the result is already in Redis:
  - If **yes** → return the cached result (parsed from JSON)
  - If **no** → make a prediction using the ML model
- The prediction result is cached in Redis with a **TTL (Time To Live)** of 1 hour (3600 seconds)
- The result is returned as JSON

3.2 - Caching DB Query Results

Tuesday, June 17, 2025 10:54 PM

Code:

- **get_db_connection():**
    - Opens a connection to sqlite3 database
    - **row_factory = sqlite3.Row** makes rows behave like dictionaries

App Workflow:

- Request comes in with **user_id**
- Redis is checked first
    - If **hit** → result returned from cache
    - If **miss** → query DB, store in cache, and return result
- Cache avoids repetitive DB load for same inputs

3.3 - Caching External API Call

Tuesday, June 17, 2025 11:01 PM

- This application will fetch data from a public API, cache the result in Redis, and return the cached result for repeated calls with the same input
- **API:** https://jsonplaceholder.typicode.com/posts
- Ensure to have **httpx** installed if not already

App Workflow:

- Generates a **unique cache key** for the given **post_id**
- Checks Redis if data exists
- If **found** → Returns cached data
- If **not** → Calls external API, caches result, and returns it
- Subsequent calls with the same ID hit **Redis** only

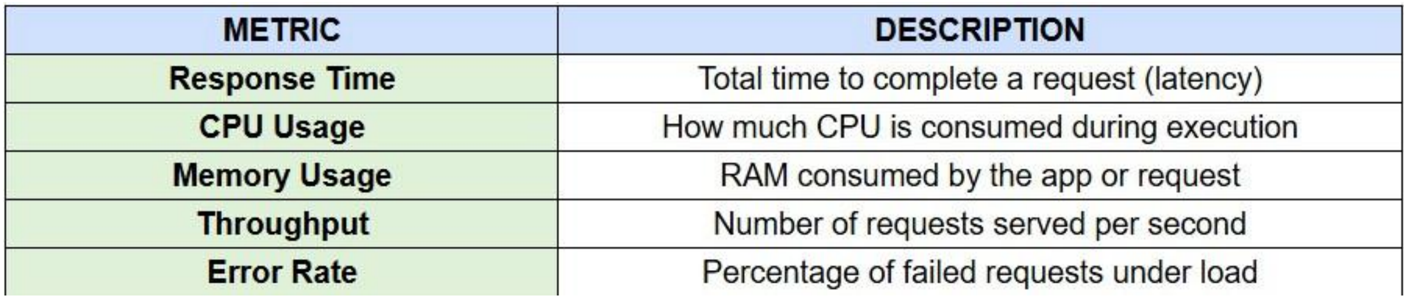
4. Profiling FastAPI Apps

Tuesday, June 17, 2025 11:17 PM

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

Key Metrics to Observe:



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

Profiling Tools:

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

4.1 - Profiling using time

Tuesday, June 17, 2025 11:39 PM

4.2 - Profiling using cProfile

Tuesday, June 17, 2025 11:40 PM

- Install **snakeviz** for interactive visualization:
  - **snakeviz** is a browser-based visualization tool for Python's **cProfile** output
  - It provides interactive graphs to see how much time each function takes and how functions call each other

- Run **snakeviz**:
  - **snakeviz profiles\<profile_name>.prof**

4.3 - Profiling using line_profiler

Friday, June 20, 2025 10:46 PM

- Install **line_profiler**
- Write code for **app.py**
- Write code for **benchmark_test.py**
- Run the profiler --> kernprof -l -v profiling_test.py
    - **-l**: line-by-line profiling
    - **-v**: verbose output

Note:

- **line_profiler** doesn't run inside an ASGI server like **uvicorn** directly; needs isolated functions for benchmarking
- **line_profiler** installs a command-line tool called **kernprof**
- Run the profiler with **kernprof**
- **@profile** is not a built-in python decorator, it tells **line_profiler** - **Profile this function line by line**

5. Benchmarking APIs

Friday, June 20, 2025 11:08 PM

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

5.1 - Advantages

Friday, June 20, 2025 11:15 PM

1. Identify Performance Bottlenecks:

- Benchmarking allows you to pinpoint **slow endpoints**, **memory-heavy operations**, and **latency-inducing logic**
    - **Are certain API routes taking longer than expected?**
    - **Is the database query slowing down the response?**
    - **Are your background tasks affecting throughput?**
- By benchmarking, you gain visibility into which part of the stack needs optimization — whether it's the application layer, the database, or external integrations

2. Assess Scalability:

- Benchmarking simulates varying loads to evaluate how well your system scales
    - **How many concurrent users can your API support?**
    - **How does the performance degrade as the load increases?**
    - **What's the tipping point before errors or timeouts occur?**
- These insights are critical when preparing for **traffic spikes**, **product launches**, or **promotional campaigns**

3. Validate Service Level Agreements (SLAs):

- SLAs often define maximum response times, uptime, and error rates
- Benchmarking helps ensure:
    - Response times remain within the acceptable threshold (e.g., under 200ms)
    - Error rates stay below critical limits under heavy usage
    - Availability is maintained across different endpoints and services
- Failing to meet SLAs can result in **penalties, lost trust** and **poor user experience**

4. Compare Different Implementations:

- When considering design or technology changes, benchmarking helps you compare performance across versions
    - Comparing REST vs. GraphQL
    - Evaluating different FastAPI configurations (sync vs async)
    - Switching databases (PostgreSQL vs MongoDB)
    - Assessing the impact of caching layers like Redis
- This ensures that new implementations offer tangible benefits and don't degrade performance

5. Optimize Cost and Resources:

- Cloud providers charge based on usage
  - Tune your application to use resources efficiently
  - Avoid over-provisioning compute instances
  - Spot memory leaks or CPU-hungry operations
- It ensures you're not overpaying for underperforming infrastructure

6. Improve End-User Experience:

- End-users expect fast and consistent performance
  - Low latency under real-world usage conditions
  - Predictable performance regardless of geography or concurrency
  - Resilience under peak demand
- This directly influences user retention, satisfaction, and product adoption

7. Prepare for Production Readiness:

- Before going live, benchmarking gives confidence that:
  - Your system is production-ready
  - It can survive worst-case scenarios (e.g., DDoS mitigation, flash sales)
  - You can confidently scale your services

5.2 - Key Metrics

Friday, June 20, 2025 11:38 PM

1. **Latency: The time taken by the API to respond to a request, typically measured in milliseconds**
   - **Importance:**
     - Indicates user experience
     - **High tail latency** often causes **performance jitter** under load
   - **Use Cases:**
     - Detect **performance regression** after code changes
     - Optimize specific endpoints for faster responsiveness

2. **Throughput:** The number of requests the API can handle per second
   - **Importance:**
     - Tells you how much load the system can sustain
     - Key for **capacity planning** and **scaling decisions**
   - **Use Cases:**
     - Compare system performance under various configuration
     - Ensure infrastructure can handle expected traffic peaks

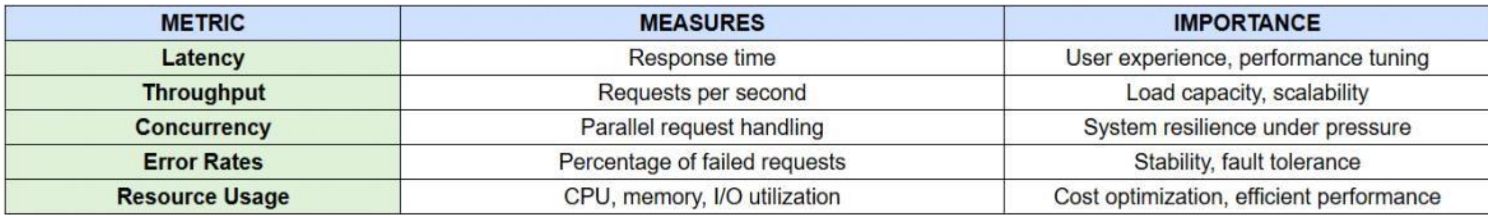
3. **Concurrency Handling:** The ability of API to process multiple simultaneous requests without degradation in performance or errors
   - **Importance:**
     - Real-world users hit APIs simultaneously
     - Poor concurrency support leads to timeouts and failed transactions
   - **Use Cases:**
     - Test **async vs sync** performance in FastAPI
     - Ensure proper **thread/worker configurations** in production environments

4. **Error Rates:** The percentage of requests that result in errors (HTTP 4xx or 5xx responses)
   - **Importance:**
     - Indicates system instability or poor error handling under load
     - Helps distinguish between **functional errors** and **load-induced failures**

- **Use Cases:**
  - Validate rate-limiting, authentication, or fail-safes
  - Ensure robust fallback mechanisms and exception handling in production

5. **Resource Usage:** The consumption of system resources (CPU, RAM, disk I/O) by API server during execution
   - **Importance:**
     - High CPU usage can indicate inefficient code or unnecessary computation
     - Memory bloat may lead to signal leaks or overuse of objects in memory
     - Important for **cloud cost optimization** and **container-based deployments**
   - **Use Cases:**
     - Optimize code paths or data processing
     - Determine resource requirements for autoscaling in **Kubernetes** or **cloud VMs**
     - Tune configurations

Summary:



METRIC	MEASURES	IMPORTANCE
Latency	Response time	User experience, performance tuning
Throughput	Requests per second	Load capacity, scalability
Concurrency	Parallel request handling	System resilience under pressure
Error Rates	Percentage of failed requests	Stability, fault tolerance
Resource Usage	CPU, memory, I/O utilization	Cost optimization, efficient performance

5.3 - Tools

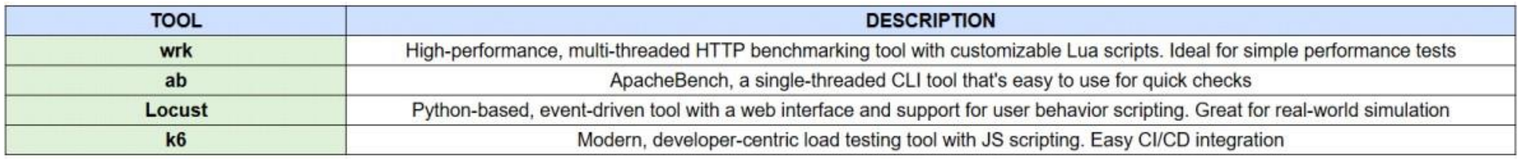
Friday, June 20, 2025 11:57 PM

1. **wrk: CLI**
   - **Pros:**
     - High performance
     - Lightweight and fast
     - Ideal for simple stress testing
   - **Cons:**
     - No detailed report
     - No scripting logic for real user scenarios

2. **ApacheBench (ab):**
   - **Pros:**
     - Very simple to use
     - Comes pre-installed on many systems
   - **Cons:**
     - Single-threaded (not suitable for high-scale)
     - Basic reporting
     - No user behavior scripting

3. **Locust: Python-based, Web UI**
   - **Pros:**
     - Web-based control panel
     - Easy scenario scripting (supports login, flows, etc.)
     - Supports distributed testing
   - **Cons:**
     - Heavier than CLI tools
     - Slightly more setup

4. **k6: Modern CLI, CI/CD Friendly**
   - **Pros:**
     - Great for automation
     - JavaScript-based scripting
     - Detailed metrics, CI/CD integration
     - Cloud options available (k6 Cloud)
   - **Cons:**
     - Slightly complex setup for beginners



TOOL	DESCRIPTION
wrk	High-performance, multi-threaded HTTP benchmarking tool with customizable Lua scripts. Ideal for simple performance tests
ab	ApacheBench, a single-threaded CLI tool that's easy to use for quick checks
Locust	Python-based, event-driven tool with a web interface and support for user behavior scripting. Great for real-world simulation
k6	Modern, developer-centric load testing tool with JS scripting. Easy CI/CD integration

5.4 - Locust Demo

Saturday, June 21, 2025 12:31 AM

- Install **locust** using **pip**

- Implement the script:
  - **main.py**
  - **locustfile.py**
    - **HttpUser:** Represents a simulated user making HTTP requests
    - **@task:** Used to mark methods as tasks that Locust will execute
    - **between:** Used to set wait time between tasks (to simulate real user behavior)

- Run **locust**:
  - Execute -- **locust / locust -f locustfile.py**
  - Open -- http://localhost:8089/

- Provide details:
  - **Number of users to simulate** - 100
  - **Spawn rate** - 10
  - **Host** - http://127.0.0.1:8000

5.5 - Best Practices

Saturday, June 21, 2025 12:00 AM

- Warm up the server before starting tests
- Test different endpoints separately
- Try **increasing concurrency** gradually (ramp-up)
- Use production-like data if possible
- Benchmark **locally** and in **staging/cloud**
- Use monitoring tools (**Grafana + Prometheus**) to correlate performance with system usage

6. Monitoring APIs

Saturday, June 21, 2025 12:56 AM

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

Saturday, June 21, 2025 10:03 AM

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

Saturday, June 21, 2025 10:21 AM

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

6.3 - Prometheus, FastAPI & Docker

Saturday, June 21, 2025   11:02 AM

File Structure:

project-folder/

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

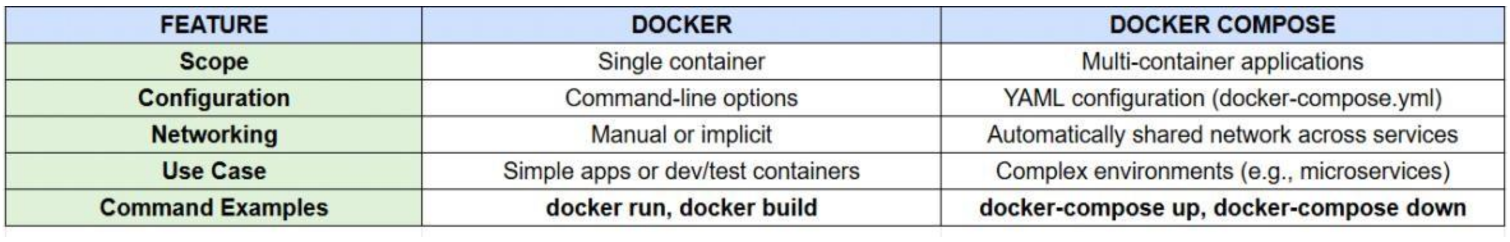
Run the app: **docker-compose up --build**

Access the Interfaces:

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

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

Saturday, June 21, 2025 11:46 AM

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

Saturday, June 21, 2025 12:10 PM

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**

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**

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

Saturday, July 5, 2025 1:03 PM

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
- **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 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

Sunday, July 6, 2025 12:24 PM

Files to setup:

- **requirements.txt**

Run the application:

- **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**

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**

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**