API

Reading Material

PW SKILLS

Topics to be Covered:

1. Introduction to API and Web API

- Definition of API (Application Programming Interface)
- Types of APIs
- Web API Concepts
- API Protocols (SOAP, REST, GraphQL)
- API Architecture Styles

2. REST API

- REST (Representational State Transfer) Principles
- HTTP Methods (GET, POST, PUT, DELETE, etc.)
- RESTful Resource Naming
- Status Codes
- API Versioning
- Authentication and Authorization in REST APIs

3. FastAPI

- Introduction to FastAPI
- Key Features of FastAPI
- Setting Up a FastAPI Project
- Request Handling and Path Parameters
- Query Parameters and Request Body
- Response Models and Status Codes
- Dependency Injection in FastAPI
- Asynchronous Programming with FastAPI

4. Difference between REST API and FastAPI

- Architectural Differences
- Performance Comparison
- Ease of Use and Development Speed
- Documentation and OpenAPI (Swagger) Support
- Type Hinting and Validation
- Asynchronous Capabilities
- Use Cases and Scenarios

Introduction to API and Web API

1. Introduction to API and Web API

Definition of API (Application Programming Interface)

- **Basic Definition:** An API (Application Programming Interface) is a set of rules and protocols that allows one software application to interact with another. Essentially, APIs enable different software components to communicate, exchange data, and execute functions, without needing to understand the internal workings of each other.

- **Analogy:** Just as human communication relies on language to exchange thoughts, APIs serve as the "language" through which software components interact. This interaction can happen between two different systems (e.g., a mobile app and a server) or within different parts of the same system.

- **Core Purpose:** The primary purpose of an API is to simplify and accelerate the development process by enabling developers to build upon existing software components. Instead of recreating functionalities, developers can use APIs to integrate or extend capabilities.

Examples of API Usage:

- **Third-Party Integration:** Social media sharing buttons on a website.
- **Internal System Integration:** A company's HR system interacting with the payroll system.
- **Service Expansion:** Adding payment gateways to an e-commerce site.

Types of APIs

1- Based on Availability:

- **Private APIs:** These are designed for use within an organization. They are often used to integrate internal systems, allowing different parts of a company's software to communicate securely and efficiently.
- **Partner APIs:** Shared externally with specific business partners, usually under contractual agreements. They enable close collaboration between companies by allowing controlled access to certain features or data.
- **Public APIs:** Also known as external or developer-facing APIs, these are available to any third-party developer. They can be free (open APIs) or require a subscription (commercial APIs). Public APIs are often used to promote brand awareness and enable broad integration with a wide range of applications.

2- Based on Use Cases:

- **Database APIs:** These allow communication between an application and a database management system, enabling operations like data retrieval, updates, and management.
- **Operating System APIs:** These APIs provide a way for applications to interact with the operating system's resources and services. Examples include Windows API and Linux API.
- **Remote APIs:** Enable interaction between applications running on different machines. They are typically used over a network, such as the internet, and include web APIs.
- **Web APIs:** The most common type, these facilitate the exchange of data and functionalities between web-based systems, following a client-server architecture.

Web API Concepts

A Web API is a specific type of API that allows for interaction between web-based systems over the internet, typically using HTTP/HTTPS protocols. It enables developers to access web services or data through a defined interface.

Client-Server Architecture: In a Web API, the client (usually a web application or mobile app) sends a request to the server, which processes the request and sends back a response. The Web API acts as the intermediary that handles this communication.

End-to-End Interaction:

- **API Call:** The client sends a request to the server, specifying an operation (e.g., retrieving data).
- **Endpoint:** The specific URL where the API call is directed.
- **Response:** The server processes the request and sends back the requested data or an error message.

API Components:

- **Calls:** Requests made by the client to the server.
- **Endpoints:** The entry points for API calls.
- **Keys:** Authentication mechanisms to ensure secure access to the API.

API Protocols (SOAP, REST, GraphQL)

1. SOAP (Simple Object Access Protocol):

- **Definition:** A protocol for exchanging structured information in web services using XML.
- **Characteristics:** It is highly standardized, supports only XML format, and is often used in enterprise applications for secure, transaction-oriented operations.
- **Usage:** Common in payment gateways, CRM solutions, and telecommunication services.

2. REST (Representational State Transfer):

- **Definition**: An architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE) for web services.
- **Characteristics:** REST is lightweight, supports multiple formats (JSON, XML, etc.), and is widely adopted for public APIs. It treats data as resources, each identified by a unique URL.
- **Usage:** REST is popular for building scalable web services, such as those provided by social media platforms, travel companies, and online marketplaces.

3. GraphQL:

- **Definition:** A query language for APIs that allows clients to specify exactly what data they need.
- **Characteristics:** Unlike REST, which returns a fixed structure, GraphQL enables clients to request precise data, making it efficient for applications with complex data requirements.
- **Usage:** Ideal for modern web and mobile apps where data needs to be fetched from multiple sources in a single request.

API Architecture Styles

1. Monolithic vs. Microservices Architecture:

- **Monolithic Architecture:** In a monolithic architecture, the entire application is built as a single unit, where APIs serve as internal connectors between different parts of the system.
- **Microservices Architecture:** In microservices architecture, the application is divided into smaller, independent services that communicate with each other via APIs. This style offers greater flexibility, scalability, and maintainability.

2. Stateful vs. Stateless APIs:

- **Stateful APIs:** The server retains the client's session information between requests, which can be useful for complex transactions but may lead to scalability issues.
- **Stateless APIs:** Each request from the client is treated independently, with no stored session on the server. RESTful APIs are typically stateless, making them more scalable and easier to manage.

3. Event-Driven APIs (Webhooks):
Webhooks are event-driven APIs that automatically trigger an action in response to specific events (e.g., a new booking in a hotel management system).

- **Characteristics:** They are lightweight and easy to set up, making them suitable for real-time updates and notifications.
- **Usage:** Often used in scenarios where real-time data push is required, such as sending automated alerts or syncing systems.

REST API:

1. REST API (Representational State Transfer)
REST (Representational State Transfer) is an architectural style that defines a set of constraints to be used for creating web services. Web services that adhere to the REST architecture are called RESTful services or REST APIs. These APIs allow different software applications to communicate over the internet, often using standard HTTP methods.

1. REST Principles:
REST APIs are based on six key principles that guide their design and implementation:

- **Statelessness:** Each client-server interaction is independent, meaning that the server does not retain any client state between requests. This allows for scalability and simplifies server design.
- **Client-Server Separation:** The client and server have distinct responsibilities. The client handles the user interface and user-related tasks, while the server handles the data storage and processing.
- **Uniform Interface:** A consistent, standardized interface that simplifies interactions between clients and servers.
- **Layered System:** The architecture allows for layers between the client and the server, such as caching and load balancing.
- **Cacheability:** Responses from the server should indicate whether they can be cached by the client to optimize performance.
- **Code on Demand (Optional):** The server can send executable code to the client to extend functionality.

2. HTTP Methods
REST APIs use standard HTTP methods to perform operations on resources. Each method corresponds to a specific action:

- **GET:** Retrieves a resource or collection of resources.
- **POST:** Creates a new resource.
- **PUT:** Updates an existing resource.
- **DELETE:** Remove a resource.
- **PATCH:** Partially updates a resource.
- **OPTIONS:** Describes the communication options for the target resource.

3. **RESTful Resource Naming**
Resource naming is a critical aspect of REST API design. Proper naming conventions ensure that APIs are intuitive and easy to use.

- **Use Nouns, Not Verbs:** Endpoints should represent resources, so use nouns (e.g., `/users` instead of `/getUsers`).
- **Hierarchical Structure:** Organize resources in a way that reflects their relationships, such as `/users/{userId}/orders`.
- **Plural Names for Collections:** Use plural nouns for endpoints that represent collections (e.g., `/products`).
- **Hyphens for Readability:** Use hyphens to separate words in URIs (e.g., `/user-profiles`).

4. Status Codes

HTTP status codes communicate the result of an API request to the client. These codes fall into different categories:

- **2xx Success:** The request was successfully received, understood, and accepted.
  - 200 OK: The request was successful.
  - 201 Created: A resource was successfully created.
- **4xx Client Errors:** The request contains bad syntax or cannot be fulfilled.
  - 400 Bad Request: The server cannot process the request due to client error.
  - 401 Unauthorized: Authentication is required and has failed or not been provided.
  - 404 Not Found: The requested resource could not be found.
- **5xx Server Errors:** The server failed to fulfill a valid request.
  - 500 Internal Server Error: The server encountered an unexpected condition.

5. API Versioning

Versioning is crucial for maintaining backward compatibility when APIs evolve. There are several strategies for versioning REST APIs:

- **URL Versioning:** Version is included in the URL (e.g., `/v1/users`).
- **Header Versioning:** The version is specified in the request header (e.g., `Accept: application/vnd.example.v1+json`).
- **Query Parameter Versioning:** The version is specified as a query parameter (e.g., `/users?version=1`).

6. Authentication and Authorization in REST APIs

Security is a fundamental concern in REST API design, especially when dealing with sensitive data.

- **Authentication:** Ensures that the client making the request is who they claim to be.
- **OAuth 2.0:** A widely-used framework that allows third-party applications to grant limited access to an HTTP service.
- **JWT (JSON Web Tokens):** A compact, URL-safe method for representing claims between two parties.
- **Authorization:** Determines what resources the authenticated client is allowed to access.
- **Encryption:** Ensures data security during transmission, typically via SSL/TLS.

FastAPI

1. Introduction to FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. It's designed to be easy to use and to help you develop robust and high-performance APIs quickly. FastAPI automatically generates interactive API documentation, making it easier for developers to understand and test APIs. It's built on top of Starlette for the web parts and Pydantic for data handling.

Key Highlights:

- FastAPI is one of the fastest Python web frameworks available, comparable to Node.js and Go.
- It supports both synchronous and asynchronous programming.
- Automatically generates OpenAPI and JSON Schema documentation.
- Type hints and Pydantic models are utilized for data validation and serialization.

2. Key Features of FastAPI

FastAPI comes with several features that set it apart from other Python web frameworks:

- **Speed:** FastAPI is built to be fast, with performance on par with Node.js and Go.
- **Ease of Use:** The framework is designed to be easy to use and learn, especially for Python developers.
- **Automatic Interactive API Documentation:** FastAPI automatically generates documentation for your APIs using Swagger UI and ReDoc.
- **Data Validation:** Using Pydantic models, FastAPI can automatically validate and parse incoming requests.
- **Asynchronous Support:** FastAPI natively supports asynchronous programming, allowing for better performance in I/O-bound applications.
- **Dependency Injection:** FastAPI includes a powerful dependency injection system, making it easier to manage and reuse dependencies.

3. Setting Up a FastAPI Project

To start a FastAPI project, follow these steps:

3.1. Install FastAPI: You can install FastAPI and an ASGI server like `uvicorn` using pip:
Code:

```
pip install fastapi uvicorn
```

3.2. Create a Basic Application: Create a Python file (e.g., `main.py`) with the following content:
code:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}
```

3.3. Run the Application: Use `uvicorn` to run the application:
code:

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

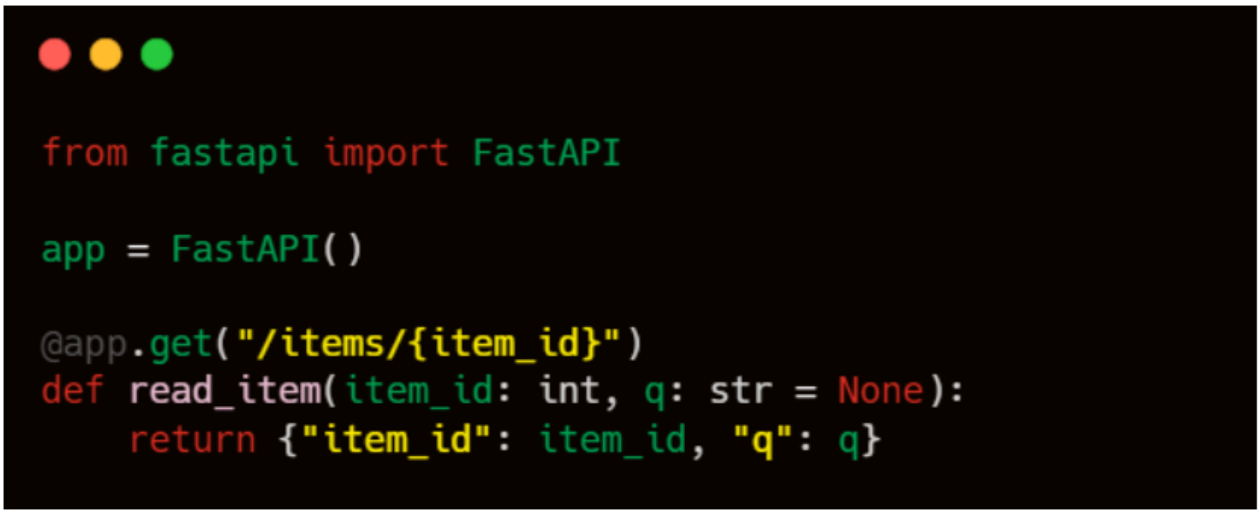
The `--reload` flag is useful during development as it automatically reloads the server on code changes.

3.4. Explore the API Documentation: Once the server is running, navigate to `http://127.0.0.1:8000/docs` to see the automatically generated Swagger UI documentation, or `http://127.0.0.1:8000/redoc` for ReDoc documentation.

4. Request Handling and Path Parameters

In FastAPI, you can define endpoints that handle HTTP requests using Python functions. Path parameters allow you to capture values from the URL path and use them in your function.

Example:



```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

In the above example:
- `{item_id}` is a path parameter. FastAPI automatically validates that `item_id` is an integer based on the type hint.
- `q` is an optional query parameter.

When you navigate to `http://127.0.0.1:8000/items/42?q=fastapi`, you'll get:
```json
{
   "item_id": 42,
   "q": "fastapi"
}
```

5. Query Parameters and Request Body

FastAPI makes it easy to work with query parameters and request bodies. Query parameters are passed in the URL, while the request body is typically sent in POST requests.

Example with Query Parameters:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/")
def read_users(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}


# Example with Request Body:

from pydantic import BaseModel
from fastapi import FastAPI

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

@app.post("/items/")
def create_item(item: Item):
    return item
```

In this example:

- The request body is validated against the `Item` Pydantic model.
- If the data doesn't match the expected format, FastAPI will return a 422 Unprocessable Entity error with details about the validation errors.

6. Response Models and Status Codes

FastAPI allows you to define response models to ensure that the data returned by your API is structured and validated. You can also specify the status code that should be returned.

Example:

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

@app.get("/items/{item_id}", response_model=Item, status_code=200)
def read_item(item_id: int):
    item = get_item_from_db(item_id)
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    return item
```

In this example:

- The `response_model` parameter ensures that the response conforms to the `Item` model.
- The `status_code` parameter explicitly sets the response status code.
- If the item is not found, a 404 Not Found error is raised with a custom error message.

7. Dependency Injection in FastAPI

FastAPI has a powerful dependency injection system that makes it easier to manage and reuse dependencies, such as database connections, authentication, and more.

Example:

```python
from fastapi import Depends, FastAPI

app = FastAPI()

def common_parameters(q: str = None, skip: int = 0, limit: int = 10):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
def read_items(commons: dict = Depends(common_parameters)):
    return commons

@app.get("/users/")
def read_users(commons: dict = Depends(common_parameters)):
    return commons
```

In this example:
- The `common_parameters` function defines common query parameters.
- The `Depends` function is used to inject these parameters into the `read_items` and `read_users` functions.

This allows you to centralize logic and reuse it across multiple endpoints.

8. Asynchronous Programming with FastAPI
FastAPI natively supports asynchronous programming, allowing you to write endpoints that perform I/O-bound operations efficiently.

Example:

```python
from fastapi import FastAPI
import asyncio

app = FastAPI()

@app.get("/async/")
async def read_async():
    await asyncio.sleep(1)
    return {"message": "This was an async operation"}
```

In this example:
- The `read_async` function is declared as `async`.
- The `await` keyword is used to pause the function execution until the `asyncio.sleep` completes.

Asynchronous functions are particularly useful when dealing with database operations, external API calls, or any other I/O-bound tasks. By leveraging async, you can handle many requests concurrently, improving the scalability and performance of your application.

The Differences between REST API and FastAPI:

1. Architectural Differences:
REST API is an architectural style for designing networked applications, while FastAPI is a modern web framework for building APIs in Python.

REST API:
- Uses standard HTTP methods (GET, POST, PUT, DELETE) to manipulate resources
- Resources are identified by unique URIs
- Promotes stateless communication and uniform interfaces
- Focuses on scalability, simplicity, and interoperability between systems

FastAPI:
- Built on top of Starlette and Pydantic
- Leverages Python's type annotations and asynchronous programming features
- Designed for high performance and easy development
- Provides a more opinionated structure compared to pure REST implementations

2. Performance Comparison:
FastAPI generally offers superior performance compared to traditional REST API implementations.

FastAPI:
- Leverages asynchronous programming, allowing for efficient handling of concurrent requests
- Built on Starlette, which is known for its high performance
- Can handle high-concurrency scenarios efficiently

REST API:
- Performance can vary depending on the specific implementation and framework used
- May not inherently support asynchronous operations, potentially leading to slower performance in high-concurrency situations

3. Ease of Use and Development Speed:

FastAPI:
- Known for its ease of use and high productivity
- Intuitive API design with clear syntax
- Automatic request body parsing and validation
- Reduced boilerplate code due to built-in features

REST API:
- Implementation complexity can vary depending on the chosen framework
- May require more manual setup for features like request validation and documentation
- Flexibility in implementation can lead to varying levels of development speed

4. Documentation and OpenAPI (Swagger) Support:

FastAPI:
- Provides automatic, interactive API documentation using Swagger UI and ReDoc
- Generates OpenAPI (formerly Swagger) specification automatically
- Documentation is always up-to-date with the code

REST API:
- Documentation often needs to be created and maintained separately
- OpenAPI support depends on the specific implementation and may require additional tools or libraries

5. Type Hinting and Validation:

FastAPI:
- Leverages Python's type annotations for request and response validation
- Automatic data validation based on type hints
- Helps catch errors early in the development process

REST API:
- Type hinting and validation depend on the specific implementation
- May require additional libraries or manual coding for robust validation

6. Asynchronous Capabilities:

FastAPI:
- Built with asynchronous programming in mind
- Supports both synchronous and asynchronous request handling
- Efficiently handles I/O-bound operations, improving overall performance

REST API:
- Asynchronous support depends on the specific implementation and framework
- Traditional REST implementations may not inherently support asynchronous operations

7. Use Cases and Scenarios:

REST API:
- Widely used across various industries and applications
- Suitable for projects requiring broad compatibility and adherence to REST principles
- Examples include:
  - E-commerce platforms (inventory management, order processing)
  - Social media aggregators
  - Financial applications (stock trading platforms)
  - IoT systems
  - Chat applications
  - Location-based services

FastAPI:
- Well-suited for modern, high-performance web applications and microservices
- Ideal for projects requiring real-time capabilities and high concurrency
- Examples include:
  - Real-time data analytics dashboards
  - IoT monitoring systems
  - Live chat applications
  - Interactive data visualization tools
  - Streaming video platforms
  - Collaborative document editing tools
  - Real-time location tracking applications

In summary, while REST API provides a flexible architectural style for building web services, FastAPI offers a more opinionated, high-performance framework specifically designed for building modern APIs in Python. FastAPI excels in areas such as performance, ease of use, automatic documentation, and built-in support for asynchronous programming, making it an attractive choice for developers building real-time and high-concurrency applications. However, REST API remains widely used and may be preferred in scenarios where broader compatibility or adherence to specific REST principles is required.