102 Workshop: Web API Development using Python (Day 2)

Presented by:
Nobel Wong
TM Technology Sdn Bhd

< / > TM Technology
Software | Cloud | Automation

Content

1. Recap
2. Persistent Storage
3. Error Handling & Status Code
4. Path Parameters, Query Parameters, Filtering
5. Day 3 Project discussion
6. Additional Topics

Recap

- Web server
- API and HTTP
- JSON - Data format
- FastAPI - Run locally, view /docs
- HTTP Methods - GET, POST, PUT, DELETE
- HTTP Status Code - 1xx, 2xx, 3xx, 4xx, 5xx
- Pydantic - Data Models for Validation
- CRUD operations for Task Manager API

Persistent Storage

- Persistent storage, also known as non-volatile storage, refers to data storage that retains information even after the power is turned off.
- Key Characteristics:
  - **Non-volatility**: The defining feature is that data remains intact without power.
  - **Long-term storage**: It's used for data that needs to be accessible for extended periods, unlike temporary storage like RAM.
  - **Data persistence**: Data is preserved even if the system is restarted, shut down, or encounters a failure.

Persistent Storage

- **Examples of Persistent Storage:**
  - **Hard Disk Drives (HDDs)**: Traditional storage devices that store data on magnetic platters.
  - **Solid State Drives (SSDs)**: More modern storage that uses flash memory for faster data access.
  - **Optical Media**: Examples include DVDs and Blu-ray discs.
  - **Cloud Storage**: Services like AWS S3, Google Cloud Storage, and Azure Blob Storage provide persistent storage for various data types.
  - **Databases**: Relational and NoSQL databases rely on persistent storage to save their data.

Persistent Storage

- **Use Cases:**
  - **Storing application data**: Ensuring that application data (user profiles, settings, etc.) is available after the application is closed and reopened.
  - **Storing system files**: Keeping the operating system and other critical system files available.
  - **Storing user files**: Allowing users to save and retrieve documents, photos, videos, etc.
  - **Database storage**: Maintaining the integrity and availability of database information.

Persistent Storage

- **Benefits**:
  - **Data Availability**: Ensures data is accessible when needed, even after system interruptions.
  - **Durability**: Provides a reliable way to store critical data that needs to be preserved.
  - **Scalability**: Persistent storage can be scaled to meet growing storage needs.

Persistent Storage

- **Common Persistent Storage**
  - **Files or blob storage**
    - GCP Cloud Storage
    - AWS S3
    - Local file system
  - **Application Data**
    - MySQL or PostgreSQL
    - MongoDB or Firebase
    - Local SQLite

Persistent Storage - JSON

- We'll use **JSON files** as our persistent storage for the rest of the workshop.
- Easy to start, no additional database installation and setup required.
- For production environment, a proper database solution is generally recommended over file-based storage due to scalability and concurrency consideration.
- Using the "json" Python module to read and write JSON files.
- Create custom functions for reading and writing.

Persistent Storage - JSON

```python
import json

DB_FILE = "data.json"

def read_data():
    try:
        with open(DB_FILE, "r") as f:
            return json.load(f)
    except FileNotFoundError:
        return []  # Return an empty list if the file doesn't exist

def write_data(data):
    with open(DB_FILE, "w") as f:
        json.dump(data, f, indent=4)
```

Task

- Update our API to use data stored in local JSON file.
- Create a reusable load_json() and save_json() function.
- Create a data.json file to store our data.
- Update all endpoints to use our local JSON file.

Error Handling

- Error handling in Python involves managing unexpected events or errors that occur during program execution, known as exceptions. This ensures that a program can gracefully recover from errors instead of crashing.
- The primary mechanism for error handling in Python is the try-except block
  - try block: This block contains the code that might potentially raise an exception
  - except block: If an exception occurs within the try block, the program execution is immediately transferred to the except block. You can specify a particular exception type to catch, or handle all.

Error Handling

- finally block (optional): This block always executes, regardless of whether an exception occurred or not. It is typically used for cleanup operations, such as closing files or releasing resources.

```python
try:
    num1 = int(input("Enter a numerator: "))
    num2 = int(input("Enter a denominator: "))
    result = num1 / num2
    print(f"The result is: {result}")
except ZeroDivisionError:
    print("Error: You cannot divide by zero!")
except ValueError:
    print("Error: Invalid input. Please enter only numbers.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
finally:
    print("Program execution finished.")
```

HTTP Status Code

- HTTP status codes are three-digit numbers returned by a server in response to a client's request (e.g., a web browser requesting a webpage). These codes communicate the outcome of the request, indicating whether it was successful, requires further action, or encountered an error.
- 1xx (Informational):
  - The server has received the request and is continuing the process. Examples include 100 Continue.
- 2xx (Successful):
  - The request was successfully received, understood, and accepted. Examples include 200 OK, 201 Created, and 204 No Content.

HTTP Status Code

- 3xx (Redirection):
  - Further action needs to be taken by the client to complete the request. Examples include 301 Moved Permanently and 302 Found.
- 4xx (Client Error):
  - The request contains an error and cannot be fulfilled by the server. Examples include 400 Bad Request, 403 Forbidden, and 404 Not Found.
- 5xx (Server Error):
  - The server failed to fulfill an apparently valid request due to an error on the server's side. Examples include 500 Internal Server Error and 503 Service Unavailable.

FastAPI Status Code Handling

- FastAPI provides several ways to handle HTTP status codes in your API responses:
  - Declaring status_code in Path Operations:
    - You can directly specify the desired HTTP status code for a successful response within the path operation decorator.

```python
from fastapi import FastAPI, status

app = FastAPI()

@app.post("/items/", status_code=status.HTTP_201_CREATED)
async def create_item(item: dict):
    return {"message": "Item created successfully", "item": item}

@app.get("/items/{item_id}", status_code=status.HTTP_200_OK)
async def read_item(item_id: int):
    # Logic to retrieve item
    return {"item_id": item_id, "name": "Example Item"}
```

FastAPI Status Code Handling

- Using HTTPException for Error Handling: For handling errors and returning appropriate client or server error status codes, HTTPException is the recommended approach.

```python
from fastapi import FastAPI, HTTPException, status

app = FastAPI()

@app.get("/users/{user_id}")
async def get_user(user_id: int):
    if user_id != 123:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"User with ID {user_id} not found"
        )
    return {"user_id": user_id, "name": "John Doe"}
```

FastAPI Status Code Handling

- Modifying Response Object Directly (Advanced):
  - For more fine-grained control, you can inject the Response object into your path operation function and set its status_code attribute. This is typically used for specific scenarios where the status code needs to be dynamically determined within the function logic.

```python
from fastapi import FastAPI, Response, status

app = FastAPI()

@app.get("/data/")
async def get_data(response: Response):
    data_exists = False # Simulate a condition
    if not data_exists:
        response.status_code = status.HTTP_204_NO_CONTENT
        return
    return {"message": "Data found"}
```

Task

- Add error handling and return correct status messages for our endpoints.
  - Status codes for all endpoints:
    - GET / - 200 OK
    - GET /tasks - 200 OK
    - GET /tasks/{task_id} - 200 OK
    - POST /tasks - 201 CREATED
    - PUT /tasks/{task_id} - 200 OK
    - DELETE /tasks/{task_id} - 200 OK
  - Proper error handling:
    - 404 Not Found: When tasks don't exist (GET, PUT, DELETE)
    - 400 Bad Request: When trying to create a task with an existing ID, or update a task to an ID that already exists

Path Parameters

- API path parameters are dynamic values embedded directly within the URL path of an API request. They are used to identify or specify a particular resource or sub-resource within an API endpoint.
- Part of the URL path:
  - Unlike query parameters, which are appended after a question mark, path parameters are integrated directly into the URL structure. They are typically denoted by curly braces {} in the API's definition.
  - Example: In /users/{id}, {id} is a path parameter.

Path Parameters

- Resource identification:
  - Path parameters are primarily used to point to a specific resource within a collection. For instance, in /products/{productId}, productId identifies a unique product.
- Required parameters:
  - Path parameters are generally considered mandatory for the request to be valid, as they are essential for identifying the target resource.
- Order matters:
  - The order of path parameters in a URL is significant, as it reflects the hierarchical structure of the resources being accessed.

Path Parameters

- Examples of usage:
  - Retrieving details of a specific user: GET /users/123
  - Accessing a particular item within a sub-collection: GET /books/{bookId}/chapters/{chapterNumber}
  - Specifying the format of a retrieved resource: GET /report.pdf
- In essence, path parameters provide a clean and intuitive way to structure API endpoints and enable precise targeting of individual resources.

Query Parameters

- API query parameters are components of a Uniform Resource Locator (URL) that allow clients to pass additional information to a web server when making a request to an API endpoint. They are used to modify the behavior of the API or to filter, sort, or paginate the data returned in the response.
- Query parameters are appended to the base URL of an API endpoint after a question mark (?). Each parameter consists of a name-value pair, separated by an equals sign (=). Multiple query parameters are separated by an ampersand (&).
    - GET /api/resource?param1=value1&param2=value2

Query Parameters: Common uses

- Filtering:
  - To retrieve a subset of data based on specific criteria (e.g., /products?category=electronics).
- Sorting:
  - To specify the order in which data should be returned (e.g., /users?sort=name&order=asc).
- Pagination:
  - To control the number of results returned and the starting point for retrieving subsequent sets of data (e.g., /items?limit=10&offset=20).
- Optional Parameters:
  - To provide additional, non-essential information to the API (e.g., /search?query=example&verbose=true).

Path vs Query Parameter

- Unlike path parameters, which are part of the URL path and identify a specific resource (e.g., /users/{id}), query parameters are typically optional and provide additional details or modifications to the request without changing the core resource being accessed.
- Path Parameters:
  - Identification of a specific resource
  - Required for resource access
  - Part of the route structure
- Query Parameters:
  - Filtering, sorting, and pagination
  - Flexible order

Task

- Add filter/search endpoint to the get tasks endpoint
  - use query parameters
  - filter by completed status
    - ?completed=true
  - auto id when creating new task
    - ?auto_id=true

API Project

- Movies/Restaurant Reviews API
  - CRUD for movies
  - CRUD for reviews
    - Each review should have an associated movies ID
- Customer Relation Management API
- Inventory Management API
- Locations API

Additional Topics

- Cloud database
- Authentication and Authorization
- OpenAPI Doc
- Deployment
- Integrating APIs - using Python

Q&A Session

Thank you

Presented by:
Nobel Wong
TM Technology Sdn Bhd