COS30049 – Computing Technology Innovation Project

Week 10 - Introduction to Back-End Development

(Lecture – 02)

Ningran Li (Icey)
ningranli@swin.edu.au

Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.

- Introduction to Back-End development
- Introduction to HTTP
- **Introduction to FastAPI**
- Introduction to Swagger & Postman
- Simple Demo

FastAPI

FastAPI framework, high performance, easy to learn, fast to code, ready for production

Test passing | coverage 100% | pypi package v0.114.0 | python 3.8 | 3.9 | 3.10 | 3.11 | 3.12

FastAPI

FastAPI Overview

What is FastAPI

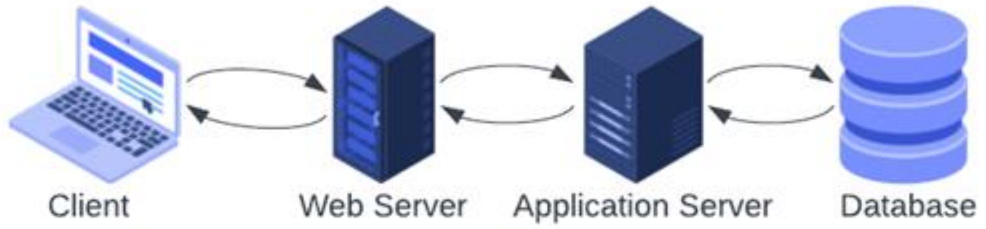
FastAPI is A modern, fast (high-performance), web framework for building APIs with Python 3.6+.

Main Components:

- **Starlette:** The lightweight ASGI framework that FastAPI is built on, providing routing, middleware, sessions, WebSockets, background tasks, and more.
- **ASGI Server:** Like Uvicorn or Hypercorn, serves the FastAPI application.
- **Path Operations:** Define API endpoints using Python functions and decorators.
- **Data Validation:** Uses **Pydantic** for parsing and validating request data with Python type hints.
- **Dependency Injection:** Built-in system to manage and inject dependencies into your functions.
- **Middleware and Background Tasks:** Allows for custom request handling layers and background processing tasks.

FastAPI Overview(Cont.)

FastAPI uses Starlette as the underlying ASGI framework. In our course, we will use **Uvicorn** as the ASGI server, corresponding to the web server in this diagram.



In the assignment for this course, we will use FastAPI to implement the application server part as shown in the diagram. Since our AI model prediction is provided as a separate service, this course will not cover the database part in the web model shown in the diagram.

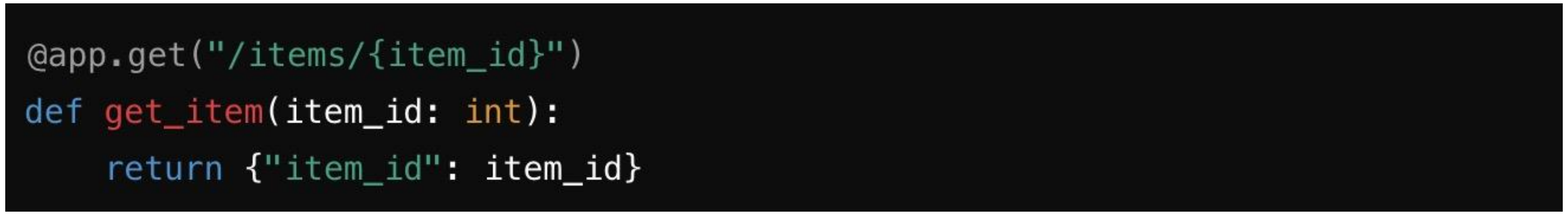
Remember:

Your success in mastering FastAPI depends on how well you leverage the **official documentation**.

Make it your go-to resource at every step of your learning journey!

Path Operations

- Functions defined in FastAPI to handle specific routes (URLs) and HTTP methods.
- Decorated with @app.get(), @app.post(), @app.put(), @app.delete(), etc., to map functions to HTTP requests.



```python
@app.get("/items/{item_id}")
def get_item(item_id: int):
    return {"item_id": item_id}
```

Defines a GET endpoint that retrieves an item based on the item_id path parameter.

Data Validation and Serialization with Pydantic

Pydantic Models:

- FastAPI uses Pydantic models to define data schemas for request and response bodies.
- Automatically validates incoming request data according to the model schema.

```python
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None
```

Data Serialization:

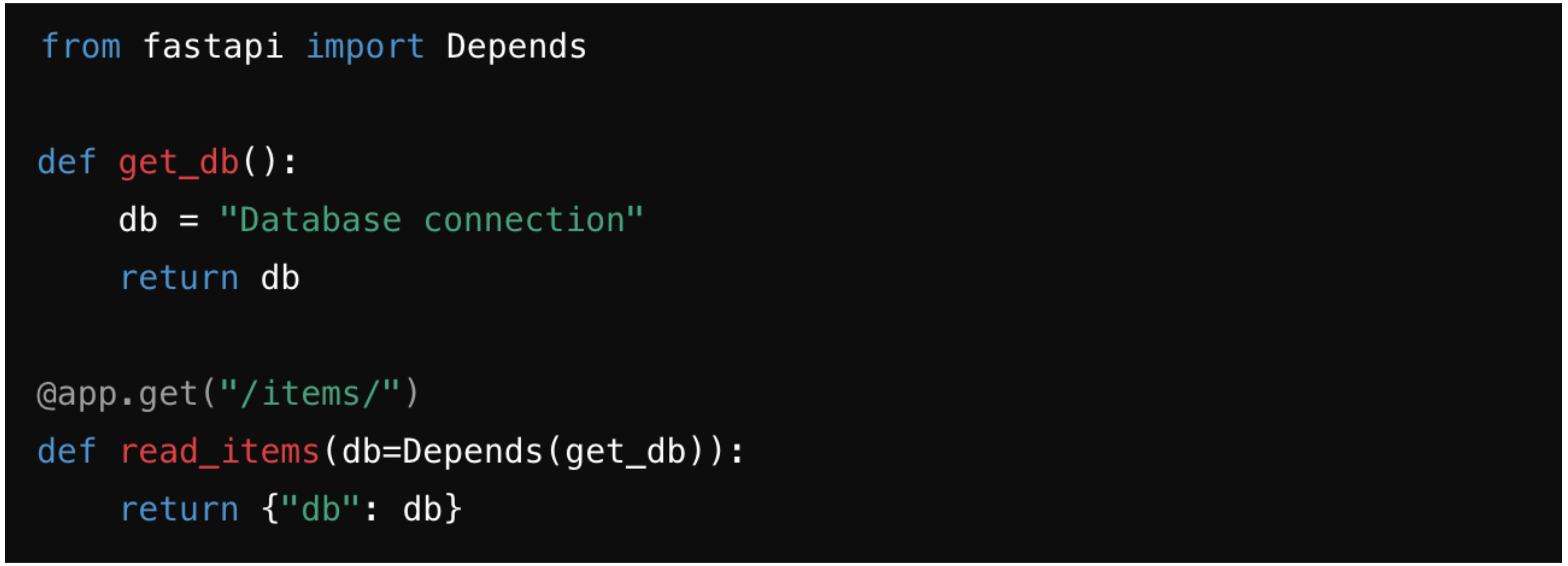
- Converts complex data types (e.g., objects) into JSON format to be sent in HTTP responses.
- Pydantic models also handle serialization seamlessly.

Dependency Injection and Middleware

Dependency Injection:

- A technique to manage and inject external dependencies (like database connections, configuration settings) into functions.
- Improves modularity and testability by separating business logic from external resources.

Example of Dependency Injection:



```python
from fastapi import Depends

def get_db():
    db = "Database connection"
    return db

@app.get("/items/")
def read_items(db=Depends(get_db)):
    return {"db": db}
```

get_db is a dependency that provides a database connection to the read_items endpoint.

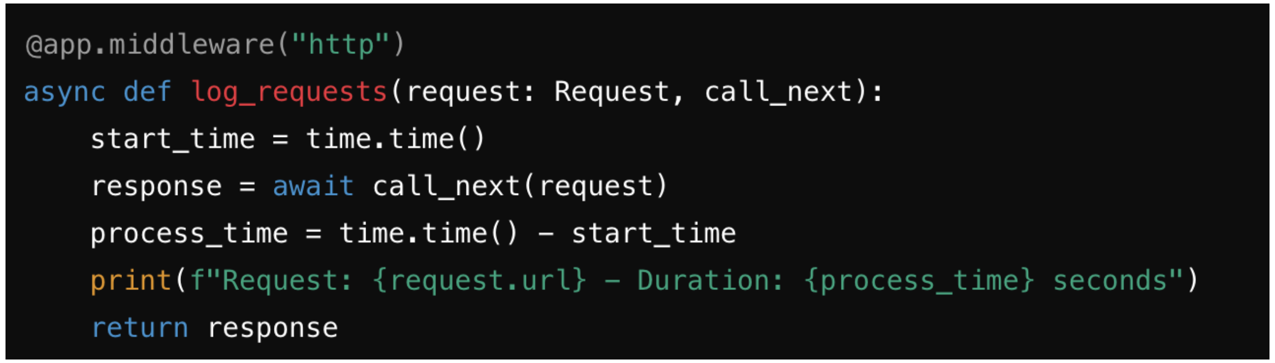
Dependency Injection and Middleware (Cont.)

Middleware:

Definition: Middleware is a function that sits between the client's request and the server's response. It processes requests before they reach the application logic and can modify responses before they are sent back to the client.

Middleware can perform tasks such as logging, handling CORS (Cross-Origin Resource Sharing), adding security headers, or processing sessions.

This middleware logs the URL of each request and the time it took to process. It demonstrates how you can use middleware to track performance and debug issues.



```python
@app.middleware("http")
async def log_requests(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    print(f"Request: {request.url} - Duration: {process_time} seconds")
    return response
```

Example of Middleware Usage

Error Handling and Exception Management

Error Handling with HTTPException:

Definition: Error handling in FastAPI involves managing exceptions that occur during request processing.

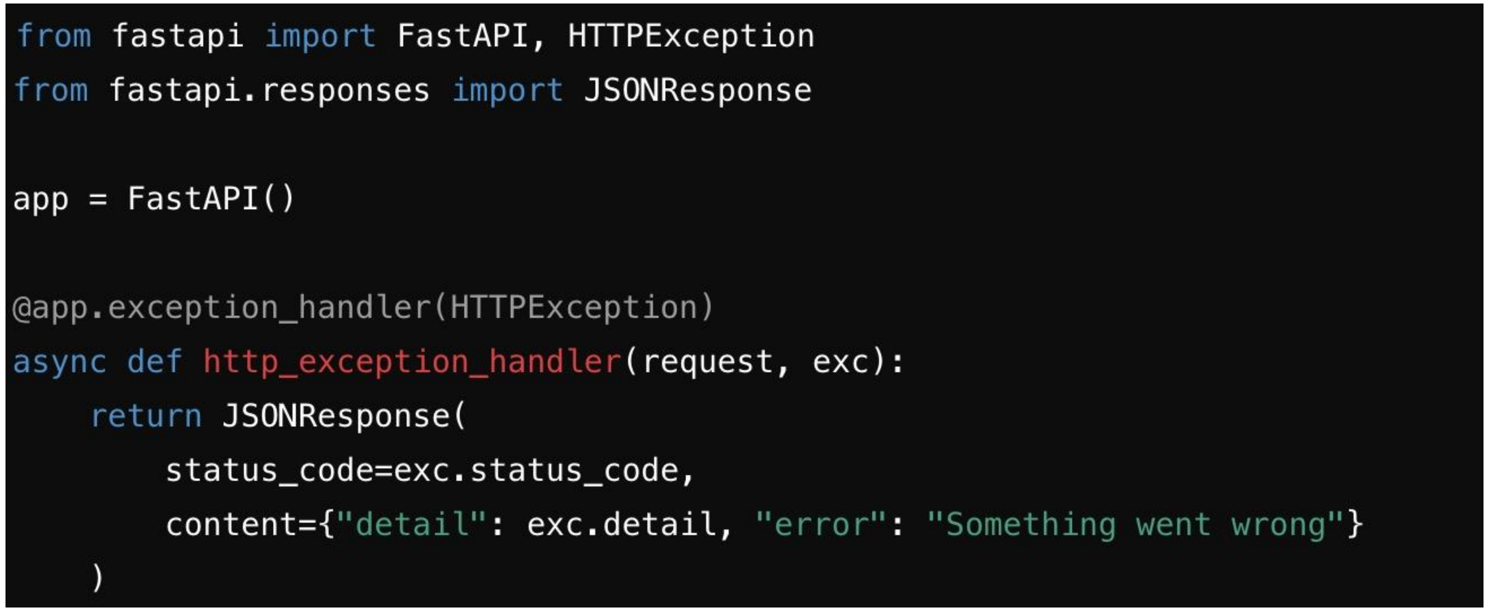
HTTPException: FastAPI uses the HTTPException class to raise HTTP errors. This class allows you to specify an HTTP status code and a detailed error message, which can be returned to the client in a standardized format. For example, you can raise a 404 Not Found error if a requested resource is not available, or a 400 Bad Request error for invalid input.

```python
from fastapi import HTTPException
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id < 0:
        raise HTTPException(status_code=400, detail="Item ID must be
positive")
    # Your logic here
    return {"item_id": item_id}
```

Error Handling and Exception Management (Cont.)

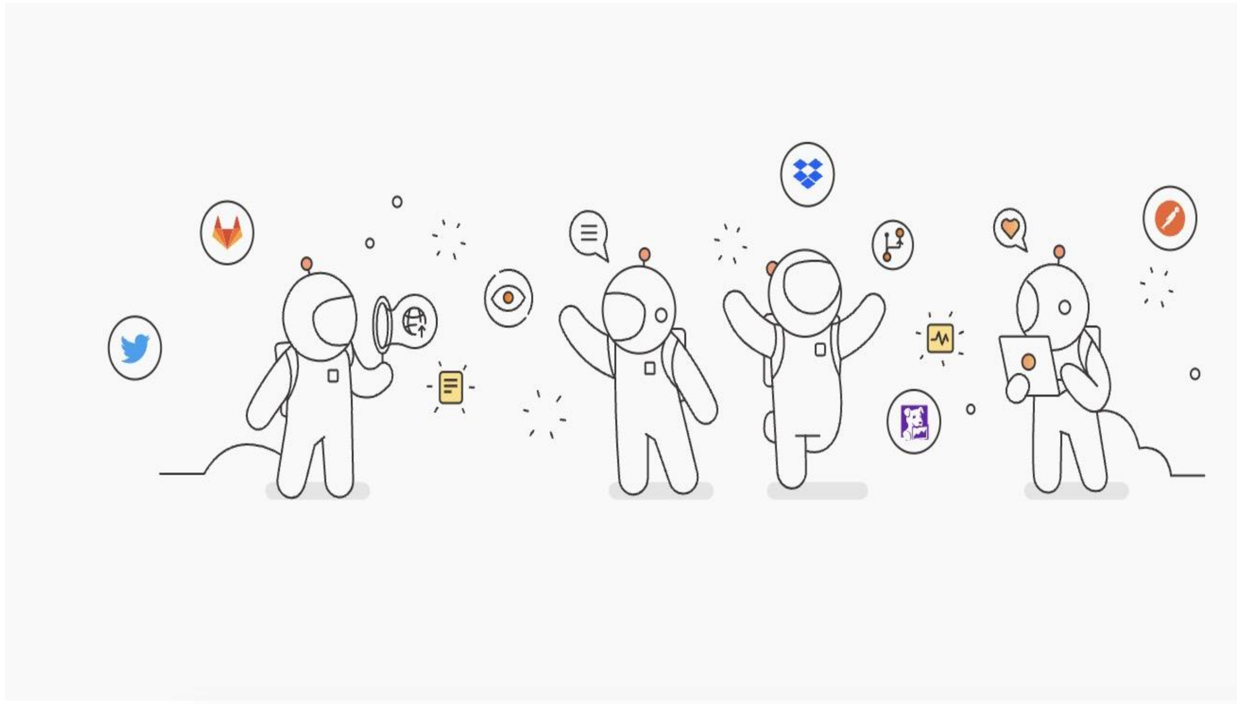
Managing Exceptions in FastAPI:

Custom Exception Handling: FastAPI allows you to define custom exception handlers for specific types of exceptions. You can create handlers to format error responses consistently across your application or to provide additional context.



```python
from fastapi import FastAPI, HTTPException
from fastapi.responses import JSONResponse

app = FastAPI()

@app.exception_handler(HTTPException)
async def http_exception_handler(request, exc):
    return JSONResponse(
        status_code=exc.status_code,
        content={"detail": exc.detail, "error": "Something went wrong"}
    )
```

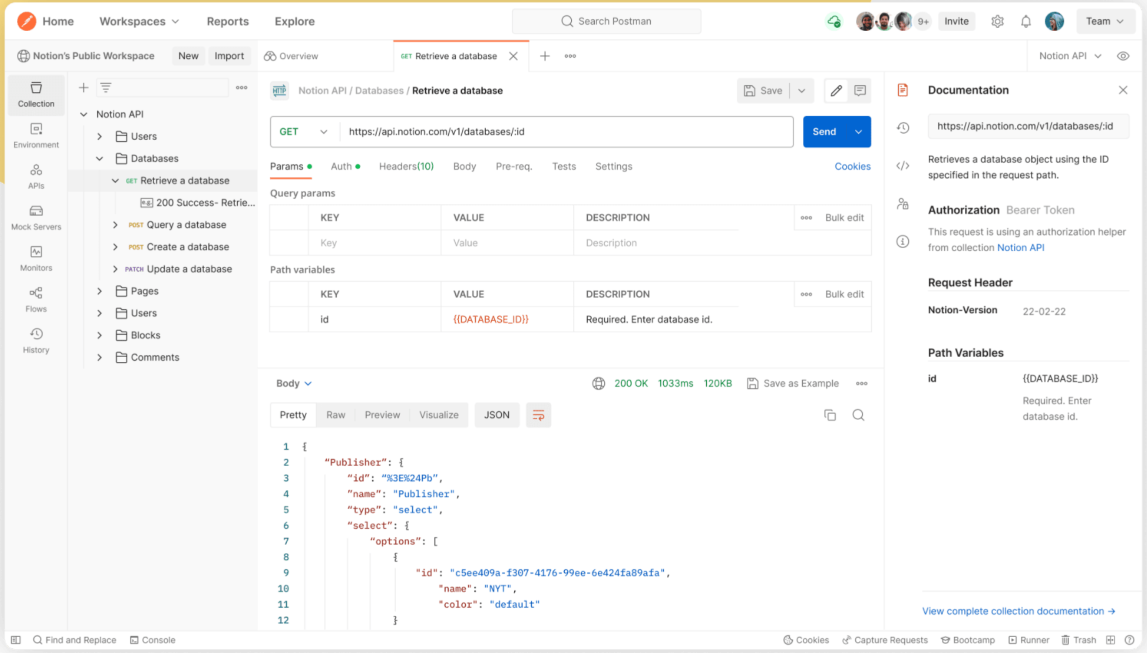
- Introduction to Back-End development
- Introduction to HTTP
- Introduction to FastAPI
- **Introduction to Swagger & Postman**
- Simple Demo



Postman

Introduction to Postman

Definition: Postman is a popular API testing tool that allows developers to create, test, and document APIs. It provides a user-friendly interface to interact with your APIs, send various types of HTTP requests, and view responses.



GET https://api.notion.com/v1/databases/:id

Query params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Path variables

KEY	VALUE	DESCRIPTION
id	{{DATABASE_ID}}	Required. Enter database id.

200 OK 1033ms 120KB

```
{
  "Publisher": {
    "id": "%3E%24Pb",
    "name": "Publisher",
    "type": "select",
    "select": {
      "options": [
        {
          "id": "c5ee409a-f307-4176-99ee-6e424fa89afa",
          "name": "NYT",
          "color": "default"
        }
```

Documentation

https://api.notion.com/v1/databases/:id

Retrieves a database object using the ID specified in the request path.

Authorization Bearer Token

This request is using an authorization helper from collection Notion API

Request Header

Notion-Version	22-02-22

Path Variables

id	{{DATABASE_ID}} Required. Enter database id.

Postman

Introduction to Postman (Cont.)

Basic Workflow:

Creating a Request:

- **Step 1:** Open Postman and click on the "New" button or use the "+" tab to create a new request.
- **Step 2:** Select the HTTP method (GET, POST, etc.) from the dropdown menu.
- **Step 3:** Enter the URL of the API endpoint you want to test.
- **Step 4:** Add any necessary headers, parameters, or request body data.
- **Step 5:** Click "Send" to execute the request and view the response.

Viewing Responses:

- **Response Tab:** Examine the status code, response time, headers, and body content in various formats (JSON, HTML, XML).
- **Pre-request and Test Scripts:** Write JavaScript code to run before sending the request or after receiving the response to perform automated testing or data manipulation.

Example: Creating a POST Request in Postman

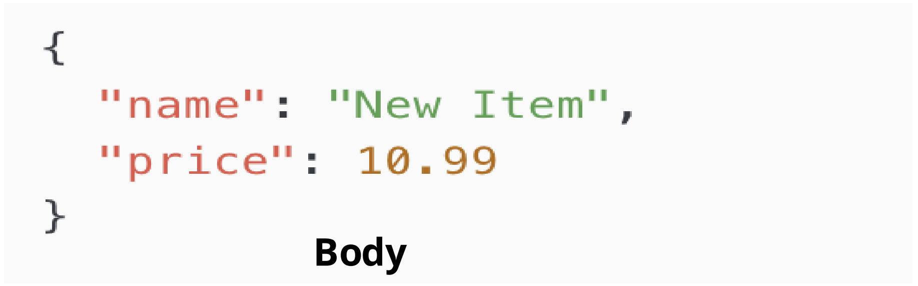
Objective: Add a new item to the API.

Steps:

1. Open Postman and click on the "New" button.
2. Select "POST" from the HTTP method dropdown.
3. Enter the URL of the API endpoint. For example, https://api.example.com/items.
4. Go to the "Body" tab and select "raw" and then "JSON".
5. Add the request body data with the new item's details.
6. Click "Send" to execute the request.

What you might see in Postman:

- **URL:** https://api.example.com/items
- **Method:** POST
- **Headers:** Content-Type: application/json
- **Body:**
- **Response:** A confirmation message and the newly created item's details in JSON format.

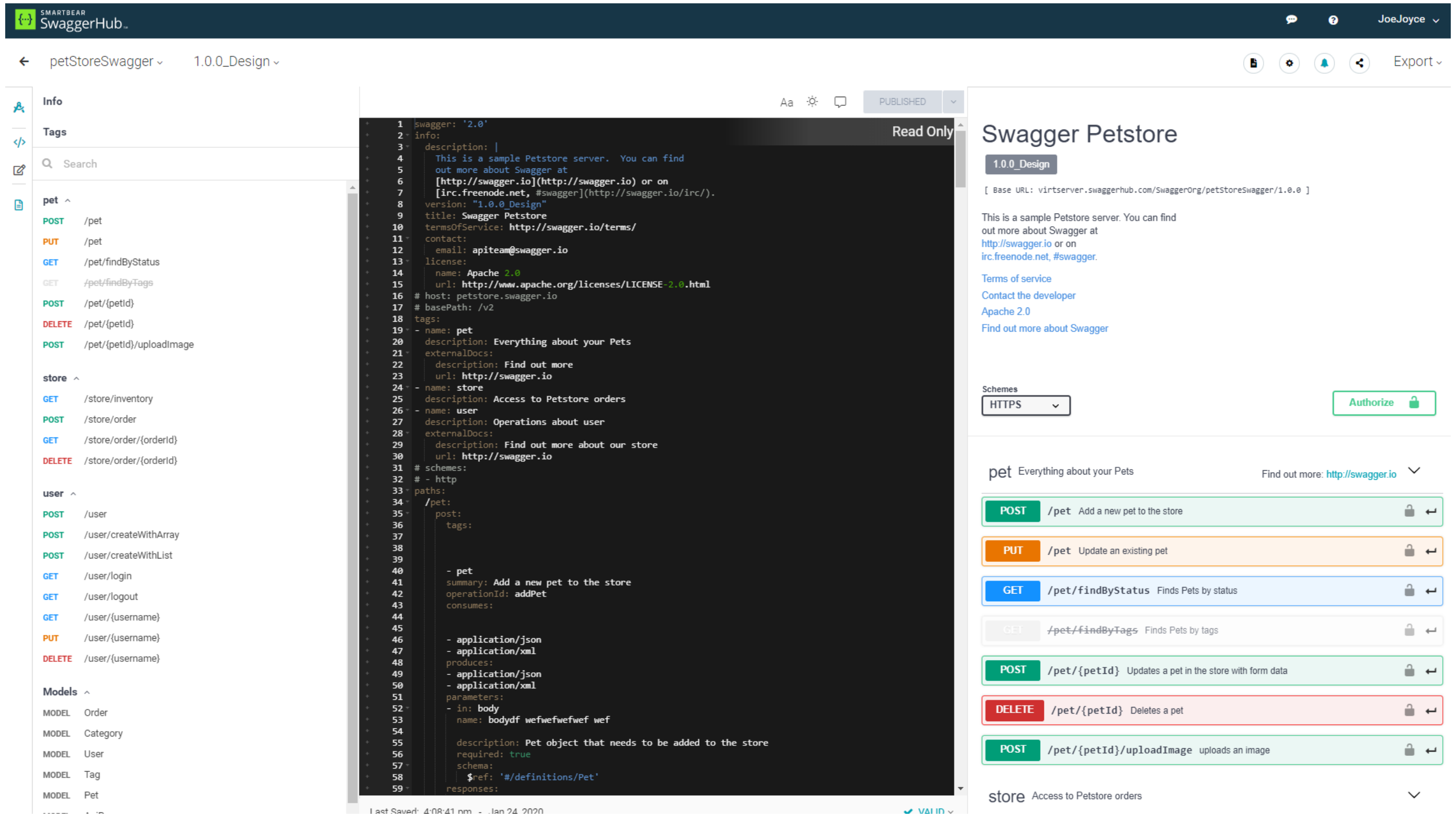


```
{
  "name": "New Item",
  "price": 10.99
}
```

Body

Introduction to Swagger

Swagger, now known as OpenAPI, is primarily a tool for API documentation and design rather than testing. However, it provide some features that can be used to interact with and test APIs.



pet
POST /pet
PUT /pet
GET /pet/findByStatus
GET ~~/pet/findByTags~~
POST /pet/{petId}
DELETE /pet/{petId}
POST /pet/{petId}/uploadImage

store
GET /store/inventory
POST /store/order
GET /store/order/{orderId}
DELETE /store/order/{orderId}

user
POST /user
POST /user/createWithArray
POST /user/createWithList
GET /user/login
GET /user/logout
GET /user/{username}
PUT /user/{username}
DELETE /user/{username}

Models
MODEL Order
MODEL Category
MODEL User
MODEL Tag
MODEL Pet

```yaml
swagger: '2.0'
info:
  description: |
    This is a sample Petstore server.  You can find
    out more about Swagger at
    [http://swagger.io](http://swagger.io) or on
    [irc.freenode.net, #swagger](http://swagger.io/irc/).
  version: "1.0.0_Design"
  title: Swagger Petstore
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
# host: petstore.swagger.io
# basePath: /v2
tags:
- name: pet
  description: Everything about your Pets
  externalDocs:
    description: Find out more
    url: http://swagger.io
- name: store
  description: Access to Petstore orders
- name: user
  description: Operations about user
  externalDocs:
    description: Find out more about our store
    url: http://swagger.io
# schemes:
# - http
paths:
  /pet:
    post:
      tags:



      - pet
      summary: Add a new pet to the store
      operationId: addPet
      consumes:


      - application/json
      - application/xml
      produces:
      - application/json
      - application/xml
      parameters:
      - in: body
        name: bodydf wefwefwefwef wef

        description: Pet object that needs to be added to the store
        required: true
        schema:
          $ref: '#/definitions/Pet'
      responses:
```

Last Saved: 4:08:41 pm - Jan 24, 2020 ✓ VALID

Swagger Petstore
1.0.0_Design
[Base URL: virtserver.swaggerhub.com/SwaggerOrg/petStoreSwagger/1.0.0]

This is a sample Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger.

Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger

Schemes: HTTPS
Authorize

pet Everything about your Pets — Find out more: http://swagger.io

POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status
GET ~~/pet/findByTags~~ Finds Pets by tags
POST /pet/{petId} Updates a pet in the store with form data
DELETE /pet/{petId} Deletes a pet
POST /pet/{petId}/uploadImage uploads an image

store Access to Petstore orders

API Request Execution in Swagger

Execute Endpoints: In Swagger UI, each endpoint in the API documentation includes an option to "Try it out." This feature lets you input parameters, headers, and request bodies, and then execute the request to see the response.

How to Try it Out: simply click on the 'Try it out' button next to an endpoint, fill in any required data, and then click 'Execute.'

View Responses: You can inspect the response status, headers, and body directly in the Swagger UI, which helps in understanding how the API behaves.

Integration with Testing

Test Automation: The OpenAPI specification (formerly Swagger) can be used to generate API client code and test cases. Tools like Postman, RestAssured, and various testing frameworks support OpenAPI specifications for automated API testing.

How it Works: When you have an OpenAPI specification file, these tools can read the file to understand the structure of your API. They can then use this information to generate client code that makes requests to your API or to create test cases that validate the API's responses.

Contract Testing: Using the OpenAPI specification as a contract, you can perform contract testing to ensure that the API implementation adheres to the documented specification.

- Introduction to Back-End development
- Introduction to HTTP
- Introduction to FastAPI
- Introduction to Swagger & Postman
- Simple Demo

[Simple Demo]

SWINBURNE UNIVERSITY OF TECHNOLOGY

Demo Overview

In this demo, we'll create a simple FastAPI application server that includes:

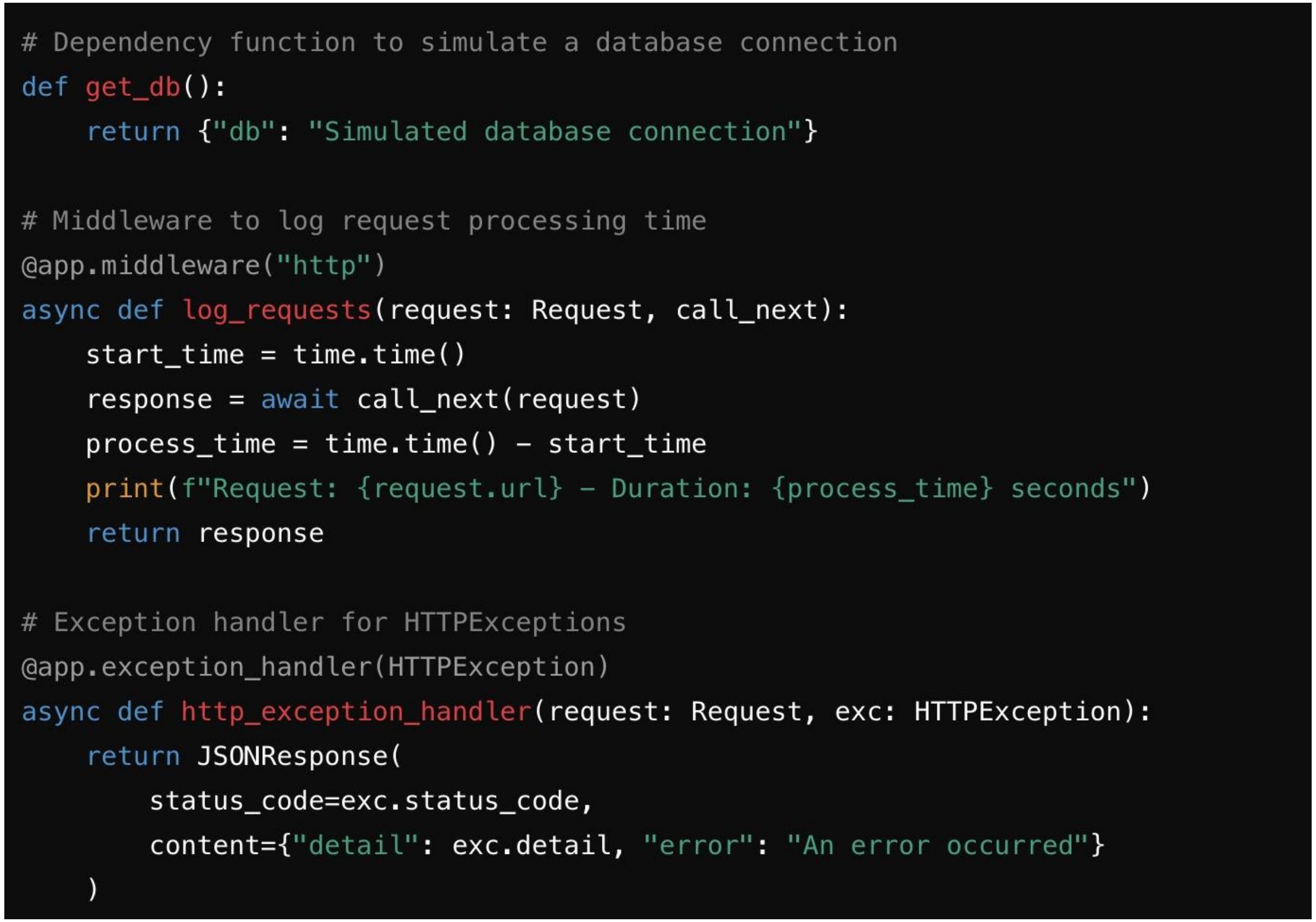
- Path Operations
- Dependency Injection
- Middleware
- Exception Handling
- API Testing with Postman

1. Setup: Installing FastAPI and Uvicorn

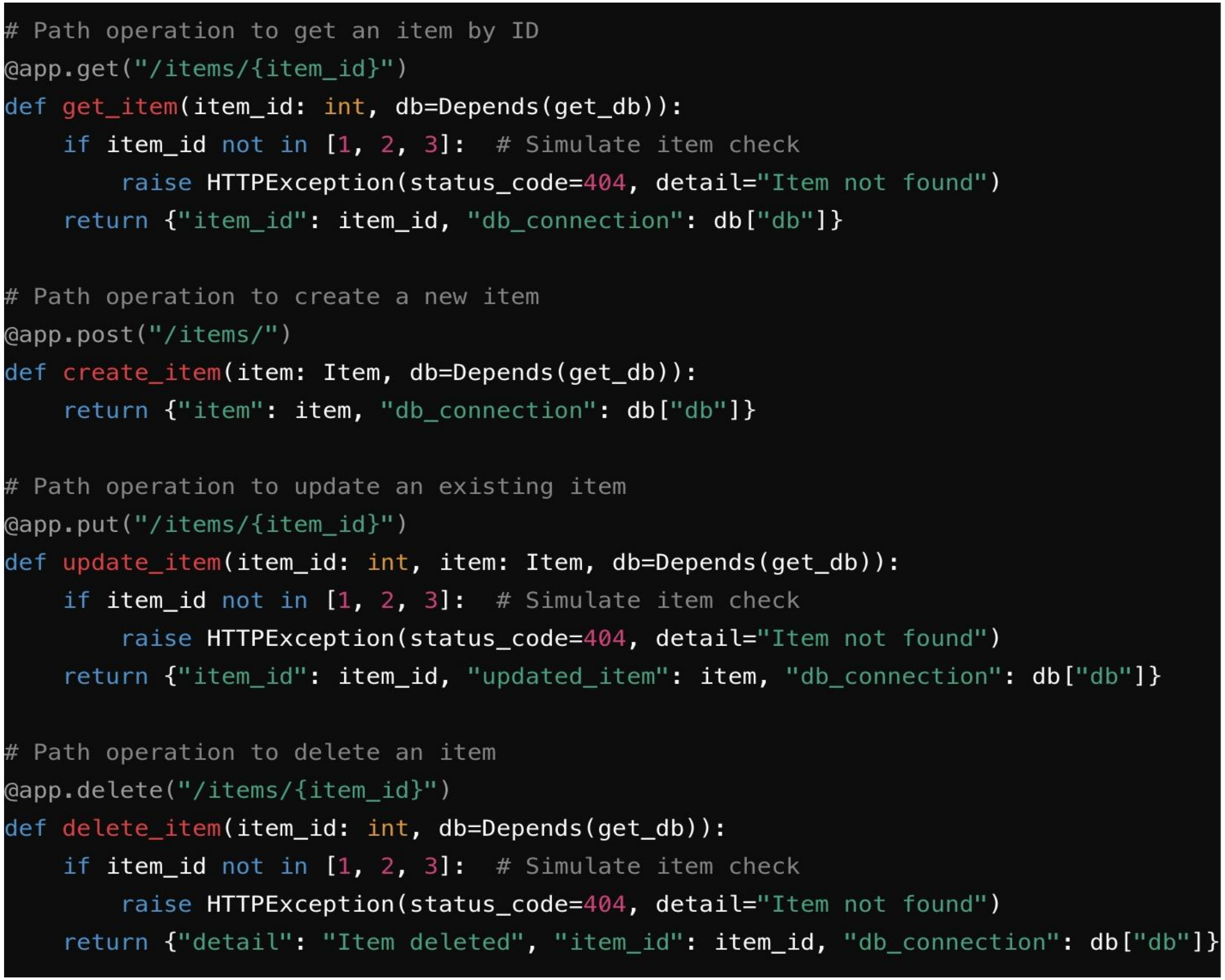
```
pip install fastapi uvicorn
```

2. FastAPI Application Code

```python
from fastapi import FastAPI, HTTPException, Depends, Request
from pydantic import BaseModel
from fastapi.responses import JSONResponse
import time

# Create FastAPI instance
app = FastAPI()

# Define a Pydantic model for data validation
class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None
```



```python
# Dependency function to simulate a database connection
def get_db():
    return {"db": "Simulated database connection"}

# Middleware to log request processing time
@app.middleware("http")
async def log_requests(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    print(f"Request: {request.url} - Duration: {process_time} seconds")
    return response

# Exception handler for HTTPExceptions
@app.exception_handler(HTTPException)
async def http_exception_handler(request: Request, exc: HTTPException):
    return JSONResponse(
        status_code=exc.status_code,
        content={"detail": exc.detail, "error": "An error occurred"}
    )
```



```python
# Path operation to get an item by ID
@app.get("/items/{item_id}")
def get_item(item_id: int, db=Depends(get_db)):
    if item_id not in [1, 2, 3]:  # Simulate item check
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id, "db_connection": db["db"]}

# Path operation to create a new item
@app.post("/items/")
def create_item(item: Item, db=Depends(get_db)):
    return {"item": item, "db_connection": db["db"]}

# Path operation to update an existing item
@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item, db=Depends(get_db)):
    if item_id not in [1, 2, 3]:  # Simulate item check
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id, "updated_item": item, "db_connection": db["db"]}

# Path operation to delete an item
@app.delete("/items/{item_id}")
def delete_item(item_id: int, db=Depends(get_db)):
    if item_id not in [1, 2, 3]:  # Simulate item check
        raise HTTPException(status_code=404, detail="Item not found")
    return {"detail": "Item deleted", "item_id": item_id, "db_connection": db["db"]}
```

3. Running the Application

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

main refers to the filename main.py.
app is the FastAPI instance.

After running the command, you should see something like this:

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to
quit)
INFO:     Started reloader process [28720]
INFO:     Started server process [28722]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

4. Testing with Postman

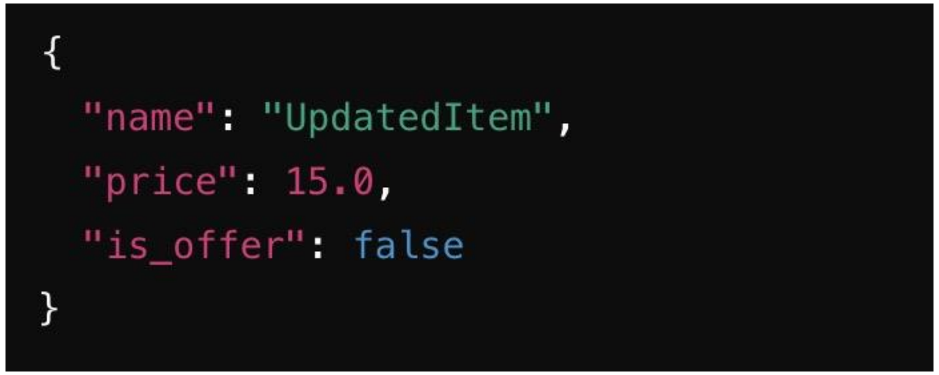
1.Open Postman:
Create a new request for each of the following endpoints.

2.GET Request to Retrieve an Item:

URL: http://127.0.0.1:8000/items/1
Method: GET
Expected Response: { "item_id": 1, "db_connection": "Simulated database connection" }

3.POST Request to Create an Item:

URL: http://127.0.0.1:8000/items/
Method: POST
Body (JSON):



```
{
  "name": "UpdatedItem",
  "price": 15.0,
  "is_offer": false
}
```

Expected Response: { "item_id": 1, "updated_item": { "name": "UpdatedItem", "price": 15.0, "is_offer": false }, "db_connection": "Simulated database connection" }

5. DELETE Request to Delete an Item:

URL: http://127.0.0.1:8000/items/1

Method: DELETE

Expected Response:

```json
{
  "detail": "Item deleted",
  "item_id": 1,
  "db_connection": "Simulated database connection"
}
```

Thank you