ASSIGNMENT-15

Task 1 – Student Records API

Task:

Use AI to build a RESTful API for managing student records.

Instructions:

- Endpoints required:
  - GET /students → List all students
  - POST /students → Add a new student
  - PUT /students/{id} → Update student details
  - DELETE /students/{id} → Delete a student record
- Use an **in-memory data structure (list or dictionary)** to store records.
- Ensure API responses are in **JSON format**.

Expected Output:

- Working API with CRUD functionality for student records.

CODE AND OUTPUT

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

# In-memory data structure to store student records
students = {}

@app.route('/students', methods=['GET'])
def get_students():
    """Get all student records."""
    return jsonify(list(students.values()))

@app.route('/students/<int:student_id>', methods=['GET'])
def get_student(student_id):
    """Get a specific student record by ID."""
    student = students.get(student_id)
    if student:
        return jsonify(student)
    return jsonify({'message': 'Student not found'}), 404

@app.route('/students', methods=['POST'])
def create_student():
    """Create a new student record."""
    data = request.json
    if not data or 'id' not in data or 'name' not in data:
        return jsonify({'message': 'Invalid student data'}), 400

    student_id = data['id']
    if student_id in students:
        return jsonify({'message': 'Student with this ID already exists'}), 409

    students[student_id] = data
    return jsonify({'message': 'Student created successfully', 'student': data}), 201
```

```
    students[student_id] = data
    return jsonify({'message': 'Student created successfully', 'student': data}), 201

@app.route('/students/<int:student_id>', methods=['PUT'])
def update_student(student_id):
    """Update an existing student record."""
    student = students.get(student_id)
    if not student:
        return jsonify({'message': 'Student not found'}), 404

    data = request.json
    if not data:
        return jsonify({'message': 'No update data provided'}), 400

    students[student_id].update(data)
    return jsonify({'message': 'Student updated successfully', 'student': students[student_id]})

@app.route('/students/<int:student_id>', methods=['DELETE'])
def delete_student(student_id):
    """Delete a student record by ID."""
    if student_id in students:
        del students[student_id]
        return jsonify({'message': 'Student deleted successfully'})
    return jsonify({'message': 'Student not found'}), 404

# To run the Flask application, you can use the following code in a separate cell
# if __name__ == '__main__':
#     app.run(debug=True)

# Note: Running Flask directly in a Colab cell can be tricky for persistent APIs.
# For a long-running API, consider using tools like ngrok or deploying to a platform.
```

```
[1]
!pip install Flask

Requirement already satisfied: Flask in /usr/local/lib/python3.12/dist-packages (3.1.2)
Requirement already satisfied: blinker>=1.9.0 in /usr/local/lib/python3.12/dist-packages (from Flask) (1.9.0)
Requirement already satisfied: click>=8.1.3 in /usr/local/lib/python3.12/dist-packages (from Flask) (8.3.0)
Requirement already satisfied: itsdangerous>=2.2.0 in /usr/local/lib/python3.12/dist-packages (from Flask) (2.2.0)
Requirement already satisfied: jinja2>=3.1.2 in /usr/local/lib/python3.12/dist-packages (from Flask) (3.1.6)
Requirement already satisfied: markupsafe>=2.1.1 in /usr/local/lib/python3.12/dist-packages (from Flask) (3.0.3)
Requirement already satisfied: werkzeug>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from Flask) (3.1.3)
```

EXPLANATION

1. **from flask import Flask, request, jsonify**: This line imports necessary components from the Flask library.
    - Flask: The main class for creating your web application.
    - request: An object that provides access to incoming request data (like JSON payloads).
    - jsonify: A helper function to return JSON responses from your API endpoints.
2. **app = Flask(_name_)**: This creates an instance of the Flask application.
3. **students = {}**: This initializes an empty Python dictionary named students. This dictionary serves as the in-memory data structure to store the student records. The keys of the dictionary will be the student IDs, and the values will be the student data (also likely dictionaries).

4. **@app.route('/students', methods=['GET'])**: This is a decorator that defines an endpoint for handling GET requests to the /students URL.
    - **def get_students():**: This function is executed when a GET request is made to /students. It returns a JSON array containing all student records stored in the students dictionary.
5. **@app.route('/students/<int:student_id>', methods=['GET'])**: This defines an endpoint for GET requests to /students/<student_id>, where <student_id> is expected to be an integer
    - **def get_student(student_id):**: This function retrieves a specific student record based on the provided student_id. If the student is found, it returns the student data as JSON; otherwise, it returns a "Student not found" message with a 404 status code.
6. **@app.route('/students', methods=['POST'])**: This defines an endpoint for handling POST requests to the /students URL.
    - **def create_student():**: This function handles the creation of a new student record. It expects the student data in JSON format in the request body. It checks for valid data (including an id and name), ensures the student ID doesn't already exist, adds the new student to the students dictionary, and returns a success message with the created student data and a 201 status code. It returns error messages with 400 or 409 status codes for invalid or duplicate data.
7. **@app.route('/students/<int:student_id>', methods=['PUT'])**: This defines an endpoint for handling PUT requests to /students/<student_id>.
    - **def update_student(student_id):**: This function updates an existing student record based on the student_id. It retrieves the student, updates their record with the data provided in the request body, and returns the updated student data. It returns a 404 if the student is not found.
8. **@app.route('/students/<int:student_id>', methods=['DELETE'])**: This defines an endpoint for handling DELETE requests to /students/<student_id>.

- **def delete_student(student_id):**: This function deletes a student record based on the student_id. If the student is found and deleted, it returns a success message; otherwise, it returns a 404.

In essence, this code sets up a simple API with endpoints to perform the basic CRUD (Create, Read, Update, Delete) operations on student data stored temporarily in the students dictionary while the Flask application is running.

Task 2:– Library Book Management API

Task: Develop a RESTful API to handle library books.

Instructions:

- Endpoints required:
  - GET /books → Retrieve all books
  - POST /books → Add a new book
  - GET /books/{id} → Get details of a specific book
  - PATCH /books/{id} → Update partial book details (e.g., availability)
  - DELETE /books/{id} → Remove a book
- Implement error handling for invalid requests.

Expected Output:

- Functional API with CRUD + partial updates.

```python
from flask import Flask, request, jsonify, abort

app = Flask(__name__)

# In-memory data structure to store book records
books = {}
next_book_id = 1 # To generate unique IDs

@app.route('/books', methods=['GET'])
def get_books():
    """Get all book records."""
    return jsonify(list(books.values()))

@app.route('/books/<int:book_id>', methods=['GET'])
def get_book(book_id):
    """Get a specific book record by ID."""
    book = books.get(book_id)
    if book:
        return jsonify(book)
    abort(404, description="Book not found")

@app.route('/books', methods=['POST'])
def create_book():
    """Create a new book record."""
    global next_book_id
    data = request.json
    if not data or 'title' not in data or 'author' not in data:
        abort(400, description="Invalid book data. 'title' and 'author' are required.")

    book_id = next_book_id
    next_book_id += 1
```

```
    new_book = {
        'id': book_id,
        'title': data['title'],
        'author': data['author'],
        'isbn': data.get('isbn'), # Optional field
        'published_year': data.get('published_year') # Optional field
    }
    books[book_id] = new_book
    return jsonify({'message': 'Book created successfully', 'book': new_book}), 201

@app.route('/books/<int:book_id>', methods=['PATCH'])
def update_book(book_id):
    """Partially update an existing book record."""
    book = books.get(book_id)
    if not book:
        abort(404, description="Book not found")

    data = request.json
    if not data:
        abort(400, description="No update data provided")

    # Update only the provided fields
    for key, value in data.items():
        if key in book: # Only update existing keys
            book[key] = value

    return jsonify({'message': 'Book updated successfully', 'book': book})

@app.route('/books/<int:book_id>', methods=['DELETE'])
def delete_book(book_id):
    """Delete a book record by ID."""
    if book_id in books:
```

```
            book[key] = value

    return jsonify({'message': 'Book updated successfully', 'book': book})

@app.route('/books/<int:book_id>', methods=['DELETE'])
def delete_book(book_id):
    """Delete a book record by ID."""
    if book_id in books:
        del books[book_id]
        return jsonify({'message': 'Book deleted successfully'})
    abort(404, description="Book not found")

# Error handler for 404 Not Found
@app.errorhandler(404)
def not_found_error(error):
    return jsonify({'message': error.description}), 404

# Error handler for 400 Bad Request
@app.errorhandler(400)
def bad_request_error(error):
    return jsonify({'message': error.description}), 400

# To run the Flask application, you can use the following code in a separate cell
# if __name__ == '__main__':
#     app.run(debug=True)

# Note: Running Flask directly in a Colab cell can be tricky for persistent APIs.
# For a long-running API, consider using tools like ngrok or deploying to a platform.
```

Explanation:

Of course! The code in cell 7f9167f5 defines a Flask-based RESTful API for managing library book records. Here's a breakdown:

1. **from flask import Flask, request, jsonify, abort**: This line imports necessary components from Flask.
    - Flask: The core class for the web application.
    - request: Accesses incoming request data.
    - jsonify: Helps in returning JSON responses.
    - abort: A function to immediately stop the request with a specific HTTP status code and an optional description (used here for error handling).
2. **app = Flask(_name_)**: Creates an instance of the Flask application.
3. **books = {}**: An in-memory Python dictionary to store book records. Keys will be the book IDs, and values will be the book data.
4. **next_book_id = 1**: An integer variable to keep track of the next available unique ID for new books.
5. **@app.route('/books', methods=['GET'])**: Defines the endpoint for GET requests to /books.
    - **def get_books()**:: Returns a JSON list of all book records in the books dictionary
6. **@app.route('/books/<int:book_id>', methods=['GET'])**: Defines the endpoint for GET requests to /books/<book_id>.
    - **def get_book(book_id)**:: Retrieves a specific book by its integer ID. If found, returns the book data as JSON. If not found, it uses abort(404, description="Book not found") to return a 404 Not Found error with a custom message.
7. **@app.route('/books', methods=['POST'])**: Defines the endpoint for POST requests to /books.
    - **def create_book()**:: Handles the creation of a new book. It expects JSON data with 'title' and 'author'. It uses abort(400, ...) if required fields are missing. It generates a unique ID using next_book_id, creates a new book dictionary (including optional 'isbn' and 'published_year' fields), adds it to the books dictionary, increments next_book_id, and returns the new book data with a 201 Created status code.
8. **@app.route('/books/<int:book_id>', methods=['PATCH'])**: Defines the endpoint for PATCH requests to /books/<book_id>.
    - **def update_book(book_id)**:: Partially updates an existing book record. It retrieves the book by ID (aborting with 404 if not found). It then iterates

through the JSON data provided in the request and updates only the fields that already exist in the book record. It aborts with 400 if no update data is provided.

9. **@app.route('/books/<int:book_id>', methods=['DELETE'])**: Defines the endpoint for DELETE requests to /books/<book_id>.
    - **def delete_book(book_id):**: Deletes a book record by its ID. If the ID exists, the book is removed from the dictionary, and a success message is returned. If not found, it aborts with a 404.
10. **@app.errorhandler(404) and @app.errorhandler(400)**: These are error handlers. They catch specific HTTP errors (404 Not Found and 400 Bad Request) that were raised using abort() within the route functions and return a consistent JSON response format containing the error's description.

This code provides a functional, albeit simple, API for managing book data in memory, demonstrating how to handle different HTTP methods and implement basic error responses using Flask.

Task 3 – Employee Payroll API

Task: Create an API for managing employees and their salaries.

Instructions:

- Endpoints required:
    - GET /employees → List all employees
    - POST /employees → Add a new employee with salary details
    - PUT /employees/{id}/salary → Update salary of an employee
    - DELETE /employees/{id} → Remove employee from system
- Use AI to:
    - Suggest data model structure.
    - Add comments/docstrings for all endpoints.

Expected Output:

- API supporting salary management with clear documentation.

CODE AND OUTPUT

```python
from flask import Flask, request, jsonify, abort

app = Flask(__name__)

# Suggested Data Model Structure:
# employees = {
#     employee_id: {
#         'id': employee_id,
#         'name': 'Employee Name',
#         'department': 'Department', # Optional
#         'salary': 0.0
#     }
# }

# In-memory data structure to store employee records
employees = {}
next_employee_id = 1 # To generate unique IDs

@app.route('/employees', methods=['GET'])
def get_employees():
    """
    Get all employee records.

    Returns:
        JSON: A list of all employee records.
    """
    return jsonify(list(employees.values()))
```

```python
@app.route('/employees', methods=['POST'])
def create_employee():
    """
    Create a new employee record.

    Expects JSON data with at least 'name' and 'salary'.
    Assigns a unique ID.

    Returns:
        JSON: A success message and the created employee record.
        Status Codes: 201 (Created), 400 (Bad Request)
    """
    global next_employee_id
    data = request.json
    if not data or 'name' not in data or 'salary' not in data:
        abort(400, description="Invalid employee data. 'name' and 'salary' are required.")

    employee_id = next_employee_id
    next_employee_id += 1

    new_employee = {
        'id': employee_id,
        'name': data['name'],
        'department': data.get('department'), # Optional field
        'salary': float(data['salary'])
    }
    employees[employee_id] = new_employee
    return jsonify({'message': 'Employee created successfully', 'employee': new_employee}), 201

@app.route('/employees/<int:employee_id>/salary', methods=['PUT'])
def update_employee_salary(employee_id):
```

```python
@app.route('/employees/<int:employee_id>/salary', methods=['PUT'])
def update_employee_salary(employee_id):
    """
    Update the salary of an existing employee.

    Expects JSON data with 'salary'.

    Args:
        employee_id (int): The ID of the employee whose salary to update.

    Returns:
        JSON: A success message and the updated employee record.
        Status Codes: 200 (OK), 404 (Not Found), 400 (Bad Request)
    """
    employee = employees.get(employee_id)
    if not employee:
        abort(404, description="Employee not found")

    data = request.json
    if not data or 'salary' not in data:
        abort(400, description="Invalid update data. 'salary' is required.")

    try:
        employee['salary'] = float(data['salary'])
    except ValueError:
        abort(400, description="Invalid salary value. Must be a number.")


    return jsonify({'message': 'Employee salary updated successfully', 'employee': employee})
```

```python
@app.route('/employees/<int:employee_id>', methods=['DELETE'])
def delete_employee(employee_id):
    """
    Delete an employee record by ID.

    Args:
        employee_id (int): The ID of the employee to delete.

    Returns:
        JSON: A success message.
        Status Codes: 200 (OK), 404 (Not Found)
    """
    if employee_id in employees:
        del employees[employee_id]
        return jsonify({'message': 'Employee deleted successfully'})
    abort(404, description="Employee not found")

# Error handler for 404 Not Found
@app.errorhandler(404)
def not_found_error(error):
    return jsonify({'message': error.description}), 404

# Error handler for 400 Bad Request
@app.errorhandler(400)
def bad_request_error(error):
    return jsonify({'message': error.description}), 400

# To run the Flask application, you can use the following code in a separate cell
# if __name__ == '__main__':
#     app.run(debug=True)

# Note: Running Flask directly in a Colab cell can be tricky for persistent APIs.
# For a long-running API, consider using tools like ngrok or deploying to a platform.
```

Explanation:

Absolutely! The code in cell ea88f049 sets up a RESTful API using Flask to manage employee records, including their salaries. Here's a breakdown:

1. **from flask import Flask, request, jsonify, abort**: Imports necessary Flask components.
    - Flask: To create the web application.
    - request: To access incoming request data (like JSON).
    - jsonify: To format responses as JSON.

- abort: To easily return standard HTTP error responses.

2. **app = Flask(_name_)**: Initializes the Flask application.
3. **# Suggested Data Model Structure:**: This is a comment section suggesting a dictionary structure for storing employee data, including id, name, department (optional),

and salary.

4. **employees = {}**: An in-memory dictionary to hold the employee records. The key will be the employee ID, and the value will be the dictionary representing the employee's data.
5. **next_employee_id = 1**: An integer used to assign unique IDs to new employees.
6. **@app.route('/employees', methods=['GET'])**: Defines the endpoint for GET requests to /employees.
    - **def get_employees():**: This function retrieves and returns a JSON list of all employee records stored in the employees dictionary.
7. **@app.route('/employees', methods=['POST'])**: Defines the endpoint for POST requests to /employees.
    - **def create_employee():**: This function handles adding a new employee. It expects JSON data containing at least 'name' and 'salary'. It uses abort(400, ...) if these required fields are missing. It assigns a unique ID

using next_employee_id, creates the employee dictionary, adds it to

the employees dictionary, and returns the new employee data with a 201 Created status code.

8. **@app.route('/employees/<int:employee_id>/salary', methods=['PUT'])**: This is a specific endpoint for PUT requests to /employees/<employee_id>/salary, designed only for updating an employee's salary.
    - **def update_employee_salary(employee_id):**: This function updates the salary of a specific employee identified by employee_id. It checks if the employee exists (abort(404, ...) if not) and expects JSON data with a 'salary' field (abort(400,

...) if missing or not a valid number). If successful, it updates the employee's salary and returns the updated record.

9. **@app.route('/employees/<int:employee_id>', methods=['DELETE'])**: Defines the endpoint for DELETE requests to /employees/<employee_id>.
    - **def delete_employee(employee_id):**: This function removes an employee record based on the employee_id. If the employee is found and deleted, it returns a success message. If not found, it uses abort(404, ...) to return a 404 error.
10. **@app.errorhandler(404) and @app.errorhandler(400)**: These are error handlers that catch 404 Not Found and 400 Bad Request errors raised by the abort() function, returning a consistent JSON response format.

Task 4 – Real-Time Application: Online Food Ordering API

Scenario:
Design a simple API for an online food ordering system.

Requirements:

- Endpoints required:
    - GET /menu → List available dishes
    - POST /order → Place a new order
    - GET /order/{id} → Track order status
    - PUT /order/{id} → Update an existing order (e.g., change items)
    - DELETE /order/{id} → Cancel an order
- AI should generate:
    - REST API code
    - Suggested improvements (like authentication, pagination)

Expected Output:

- Fully working API simulating a food ordering backend.

CODE AND OUTPUT

```python
from flask import Flask, request, jsonify, abort

app = Flask(__name__)

# In-memory data structures
# A simple static menu
menu = {
    1: {"item": "Pizza", "price": 12.99},
    2: {"item": "Burger", "price": 8.50},
    3: {"item": "Pasta", "price": 10.00},
    4: {"item": "Salad", "price": 7.00}
}

# To store orders - using a simple dictionary with auto-incrementing ID
orders = {}
next_order_id = 1

@app.route('/menu', methods=['GET'])
def get_menu():
    """
    Get the list of available menu items.

    Returns:
        JSON: A list of menu items.
    """
    return jsonify(list(menu.values()))

@app.route('/order', methods=['POST'])
def create_order():
    """
    Create a new food order.

    Expects JSON data with 'items' (a list of item IDs and quantities).
    Example: {"items": [{"item_id": 1, "quantity": 2}, {"item_id": 3, "quantity": 1}]}
```

```python
    Returns:
        JSON: A success message and the created order details.
        Status Codes: 201 (Created), 400 (Bad Request)
    """
    global next_order_id
    data = request.json
    if not data or 'items' not in data or not isinstance(data['items'], list):
        abort(400, description="Invalid order data. 'items' (list) is required.")

    ordered_items = []
    total_price = 0.0

    for item_data in data['items']:
        if 'item_id' not in item_data or 'quantity' not in item_data or item_data['item_id'] not in menu or not isinstance(item_data['quantity'], int) or item_data['quantity'] <= 0:
            abort(400, description="Invalid item data in order.")

        item_id = item_data['item_id']
        quantity = item_data['quantity']
        menu_item = menu[item_id]
        ordered_items.append({
            "item_id": item_id,
            "item_name": menu_item["item"],
            "quantity": quantity,
            "price": menu_item["price"] * quantity
        })
        total_price += menu_item["price"] * quantity

    order_id = next_order_id
    next_order_id += 1

    new_order = {
        'id': order_id,
        'items': ordered_items,
        'total_price': round(total_price, 2),
        'status': 'Pending' # Initial status
    }
```

```
    orders[order_id] = new_order

    return jsonify({'message': 'Order placed successfully', 'order': new_order}), 201

@app.route('/order/<int:order_id>', methods=['GET'])
def get_order(order_id):
    """
    Get details of a specific order by ID.

    Args:
        order_id (int): The ID of the order to retrieve.

    Returns:
        JSON: The order details.
        Status Codes: 200 (OK), 404 (Not Found)
    """
    order = orders.get(order_id)
    if order:
        return jsonify(order)
    abort(404, description="Order not found")

@app.route('/order/<int:order_id>', methods=['PUT'])
def update_order(order_id):
    """
    Update an existing order.

    Expects JSON data with fields to update (e.g., 'items', 'status').
    Note: For a real system, updating order items after placing might be complex.
    This example allows updating 'status' or replacing 'items' list entirely.

    Args:
        order_id (int): The ID of the order to update.

    Returns:
        JSON: A success message and the updated order details.
        Status Codes: 200 (OK), 404 (Not Found), 400 (Bad Request)
```

```
    order = orders.get(order_id)
    if not order:
        abort(404, description="Order not found")

    data = request.json
    if not data:
        abort(400, description="No update data provided")

    # Simple update logic: allow updating 'status' or replacing 'items'
    if 'status' in data:
        order['status'] = data['status']
    if 'items' in data and isinstance(data['items'], list):
        # Re-calculate total price if items are updated
        ordered_items = []
        total_price = 0.0
        for item_data in data['items']:
            if 'item_id' not in item_data or 'quantity' not in item_data or item_data['item_id'] not in menu or not isinstance(item_data['quantity'], int) or item_data['quantity'] <= 0:
                abort(400, description="Invalid item data in update.")
            item_id = item_data['item_id']
            quantity = item_data['quantity']
            menu_item = menu[item_id]
            ordered_items.append({
                "item_id": item_id,
                "item_name": menu_item["item"],
                "quantity": quantity,
                "price": menu_item["price"] * quantity
            })
            total_price += menu_item["price"] * quantity
        order['items'] = ordered_items
        order['total_price'] = round(total_price, 2)

    return jsonify({'message': 'Order updated successfully', 'order': order})
```

```python
@app.route('/order/<int:order_id>', methods=['DELETE'])
def delete_order(order_id):
    """
    Delete an order by ID.

    Args:
        order_id (int): The ID of the order to delete.

    Returns:
        JSON: A success message.
        Status Codes: 200 (OK), 404 (Not Found)
    """
    if order_id in orders:
        del orders[order_id]
        return jsonify({'message': 'Order deleted successfully'})
    abort(404, description="Order not found")

# Error handlers for 404 Not Found and 400 Bad Request
@app.errorhandler(404)
def not_found_error(error):
    return jsonify({'message': error.description}), 404

@app.errorhandler(400)
def bad_request_error(error):
    return jsonify({'message': error.description}), 400
```

```python
# Suggested Improvements:
# - Authentication and Authorization: Secure endpoints to ensure only authorized users can perform actions.
# - Database Integration: Use a database (like SQLite, PostgreSQL, MySQL) instead of in-memory dictionaries for persistent data.
# - More Robust Error Handling: Implement specific error types and better error messages.
# - Input Validation: More comprehensive validation of incoming data (e.g., using libraries like Marshmallow).
# - Pagination: For /menu and potentially /orders if the number of items/orders grows large.
# - User Management: Functionality to manage users.
# - Payment Integration: Connect to a payment gateway.
# - Order Status Workflow: More complex status transitions (e.g., 'Preparing', 'Delivering', 'Delivered').
# - Item Availability: Track stock or availability of menu items.

# To run the Flask application, you can use the following code in a separate cell
# if __name__ == '__main__':
#     app.run(debug=True)

# Note: Running Flask directly in a Colab cell can be tricky for persistent APIs.
# For a long-running API, consider using tools like ngrok or deploying to a platform.
```

Explanation:

Certainly! The code in cell ec995a46 defines a basic RESTful API for an online food ordering system using the Flask framework. Here's a breakdown of its components:

1. **from flask import Flask, request, jsonify, abort**: Imports necessary classes and functions from Flask, including abort for error handling.
2. **app = Flask(_name_)**: Initializes the Flask application instance.
3. **menu = {...}**: This dictionary holds the static menu data. Each key is an item ID, and the value is another dictionary containing the item's item name and price.
4. **orders = {}**: This dictionary will store the order records in memory. The key will be the order ID, and the value will be a dictionary representing the order details.

5. **next_order_id = 1**: An integer to generate unique IDs for new orders.
6. **@app.route('/menu', methods=['GET'])**: Defines the endpoint for GET requests to /menu.
    - **def get_menu():**: Returns a JSON list of all items available in the menu dictionary.
7. **@app.route('/order', methods=['POST'])**: Defines the endpoint for POST requests to /order.
    - **def create_order():**: This function handles placing a new order. It expects JSON data with an 'items' key, which should be a list of dictionaries containing 'item_id' and 'quantity'. It validates the input data, calculates the total price based on the menu, assigns a unique order_id, stores the order in

the orders dictionary, and returns the order details with a 201 Created status code. It uses abort(400, ...) for various invalid input scenarios.

8. **@app.route('/order/<int:order_id>', methods=['GET'])**: Defines the endpoint for GET requests to /order/<order_id>.
    - **def get_order(order_id):**: Retrieves and returns the details of a specific order by its integer ID. If the order is not found, it uses abort(404, ...) to return a 404 Not Found error.
9. **@app.route('/order/<int:order_id>', methods=['PUT'])**: Defines the endpoint for PUT requests to /order/<order_id>.
    - **def update_order(order_id):**: This function allows updating an existing order. It checks if the order exists (abort(404, ...) if not). It expects JSON data to update fields like 'status' or replace the 'items' list. If the 'items' list is updated, it recalculates the total price. It uses abort(400, ...) for invalid update data.
10. **@app.route('/order/<int:order_id>', methods=['DELETE'])**: Defines the endpoint for DELETE requests to /order/<order_id>.

    - **def delete_order(order_id):**: Deletes an order by its ID. If the order is found, it is removed from the orders dictionary, and a success message is returned. If not found, it uses abort(404, ...) to return a 404 error.

11. **@app.errorhandler(404) and @app.errorhandler(400)**: These are custom error handlers that catch 404 and 400 errors raised by abort() and return a consistent JSON response format with the error description.
12. **# Suggested Improvements:**: This is a comment block listing potential areas for improvement in a real-world system, such as authentication, database integration, more robust error handling, and other features.

This code provides a functional blueprint for a simple food ordering API, demonstrating how to handle different request types, process order data, and implement basic error handling within a Flask application.