# ASSIGNMENT 15

## TASK 1 :

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

app = Flask(__name__)

# In-memory data structure to store student records
students = [
    {"id": 1, "name": "Alice", "age": 20, "course": "Computer Science"},
    {"id": 2, "name": "Bob", "age": 22, "course": "Mechanical Engineering"}
]

# GET /students → List all students
@app.route('/students', methods=['GET'])
def get_students():
    return jsonify({"students": students}), 200

# POST /students → Add a new student
@app.route('/students', methods=['POST'])
def add_student():
    data = request.get_json()
    if not data or not all(k in data for k in ("name", "age", "course")):
        return jsonify({"error": "Invalid data"}), 400

    new_id = max([s["id"] for s in students]) + 1 if students else 1
    new_student = {
        "id": new_id,
        "name": data["name"],
        "age": data["age"],
        "course": data["course"]
    }
    students.append(new_student)
    return jsonify({"message": "Student added successfully", "student": new_student}), 201
```

```python
# PUT /students/{id} → Update student details
@app.route('/students/<int:student_id>', methods=['PUT'])
def update_student(student_id):
    data = request.get_json()
    for student in students:
        if student["id"] == student_id:
            student.update({
                "name": data.get("name", student["name"]),
                "age": data.get("age", student["age"]),
                "course": data.get("course", student["course"])
            })
            return jsonify({"message": "Student updated successfully", "student": student}), 200
    return jsonify({"error": "Student not found"}), 404

# DELETE /students/{id} → Delete a student record
@app.route('/students/<int:student_id>', methods=['DELETE'])
def delete_student(student_id):
    global students
    students = [student for student in students if student["id"] != student_id]
    return jsonify({"message": f"Student with ID {student_id} deleted successfully"}), 200

# Run the Flask server
if __name__ == '__main__':
    app.run(debug=True)
```

## EXPLANATION :

# ASSIGNMENT 15

It supports CRUD operations — listing all students, adding a new student, updating existing student details, and deleting a student by ID.
Each endpoint returns JSON-formatted responses with success or error messages.
The API uses request validation to ensure proper input before creating or updating records.
Running the app ( `app.run(debug=True)` ) starts a local RESTful server for testing and interaction via tools like Postman or cURL.

## TASK 2 :

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

app = Flask(__name__)

# In-memory data structure to store book records
books = [
    {"id": 1, "title": "1984", "author": "George Orwell", "available": True},
    {"id": 2, "title": "To Kill a Mockingbird", "author": "Harper Lee", "available": False}
]

# GET /books → Retrieve all books
@app.route('/books', methods=['GET'])
def get_books():
    return jsonify({"books": books}), 200

# POST /books → Add a new book
@app.route('/books', methods=['POST'])
def add_book():
    data = request.get_json()
    if not data or not all(k in data for k in ("title", "author", "available")):
        return jsonify({"error": "Invalid book data"}), 400

    new_id = max([b["id"] for b in books]) + 1 if books else 1
    new_book = {
        "id": new_id,
        "title": data["title"],
        "author": data["author"],
        "available": data["available"]
    }
    books.append(new_book)
    return jsonify({"message": "Book added successfully", "book": new_book}), 201
```

# ASSIGNMENT 15

```python
# GET /books/{id} → Get details of a specific book
@app.route('/books/<int:book_id>', methods=['GET'])
def get_book(book_id):
    for book in books:
        if book["id"] == book_id:
            return jsonify({"book": book}), 200
    return jsonify({"error": "Book not found"}), 404

# PATCH /books/{id} → Update partial book details
@app.route('/books/<int:book_id>', methods=['PATCH'])
def update_book_partial(book_id):
    data = request.get_json()
    for book in books:
        if book["id"] == book_id:
            book.update({k: v for k, v in data.items() if k in book})
            return jsonify({"message": "Book details updated", "book": book}), 200
    return jsonify({"error": "Book not found"}), 404

# DELETE /books/{id} → Remove a book
@app.route('/books/<int:book_id>', methods=['DELETE'])
def delete_book(book_id):
    global books
    for book in books:
        if book["id"] == book_id:
            books = [b for b in books if b["id"] != book_id]
            return jsonify({"message": f"Book with ID {book_id} deleted successfully"}), 200
    return jsonify({"error": "Book not found"}), 404

# Error handling for unsupported methods
@app.errorhandler(405)
def method_not_allowed(e):
    return jsonify({"error": "Method not allowed"}), 405

# Error handling for invalid JSON
@app.errorhandler(400)
def bad_request(e):
    return jsonify({"error": "Bad request - invalid input"}), 400

# Run the Flask app
if __name__ == '__main__':
    app.run(debug=True)
```

# EXPLANATION :

This Flask API manages a list of books using an in-memory data structure.
It supports CRUD operations — retrieving all books, adding a new book, fetching a specific book by ID, updating part of a book's details, and deleting a book.
Each endpoint returns JSON responses with appropriate HTTP status codes.
Error handling is included for invalid input (400) and unsupported HTTP methods (405).
The app runs in debug mode, allowing live updates and detailed error messages during development.

# ASSIGNMENT 15

## TASK 3 :

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

app = Flask(__name__)

employees = [
    {"id": 1, "name": "Alice Johnson", "position": "Manager", "salary": 75000.00},
    {"id": 2, "name": "Bob Smith", "position": "Developer", "salary": 55000.00}
]

@app.route('/employees', methods=['GET'])
def get_employees():
    return jsonify({"employees": employees}), 200

@app.route('/employees', methods=['POST'])
def add_employee():
    data = request.get_json()
    if not data or not all(k in data for k in ("name", "position", "salary")):
        return jsonify({"error": "Invalid employee data"}), 400
    new_id = max([emp["id"] for emp in employees]) + 1 if employees else 1
    new_employee = {
        "id": new_id,
        "name": data["name"],
        "position": data["position"],
        "salary": data["salary"]
    }
    employees.append(new_employee)
    return jsonify({"message": "Employee added successfully", "employee": new_employee}), 201

@app.route('/employees/<int:emp_id>/salary', methods=['PUT'])
def update_salary(emp_id):
    data = request.get_json()
    if not data or "salary" not in data:
        return jsonify({"error": "Salary value required"}), 400
    for emp in employees:
        if emp["id"] == emp_id:
            emp["salary"] = data["salary"]
            return jsonify({"message": "Salary updated successfully", "employee": emp}), 200
    return jsonify({"error": "Employee not found"}), 404

@app.route('/employees/<int:emp_id>', methods=['DELETE'])
def delete_employee(emp_id):
    global employees
    for emp in employees:
        if emp["id"] == emp_id:
            employees = [e for e in employees if e["id"] != emp_id]
            return jsonify({"message": f"Employee ID {emp_id} removed successfully"}), 200
    return jsonify({"error": "Employee not found"}), 404
```

```python
@app.errorhandler(405)
def method_not_allowed(e):
    return jsonify({"error": "Method not allowed"}), 405

@app.errorhandler(400)
def bad_request(e):
    return jsonify({"error": "Bad request - invalid input"}), 400

if __name__ == '__main__':
    app.run(debug=True)
```

# ASSIGNMENT 15

## EXPLANATION :

This Flask-based REST API is designed to manage employee records using an in-memory list (no database).
The `/employees` GET endpoint returns all employee details in JSON format.
The `/employees` POST endpoint allows adding a new employee by providing name, position, and salary in the request body.
Each new employee is automatically assigned a unique ID.
The `/employees/<id>/salary` PUT endpoint updates only the salary of a specific employee based on their ID.
If the given employee ID doesn't exist, the API returns a 404 error message.
The `/employees/<id>` DELETE endpoint removes an employee record by matching the provided ID.
All responses are JSON-formatted with proper HTTP status codes for success or error conditions.
The API includes error handling for invalid input (400) and unsupported HTTP methods (405).
The `debug=True` mode allows developers to test and debug the API interactively.
This simple structure demonstrates basic CRUD (Create, Read, Update, Delete) operations in a Flask environment.
It can be extended later to connect with a real database for persistent storage.

## TASK 4 :

```python
from flask import Flask, jsonify, request
app = Flask(__name__)
# In-memory storage for menu and orders
menu = [
    {"id": 1, "name": "Margherita Pizza", "price": 250},
    {"id": 2, "name": "Veg Burger", "price": 150},
    {"id": 3, "name": "Pasta Alfredo", "price": 200}
]

orders = []


# ====================================
# GET /menu → List available dishes
# ====================================
@app.route('/menu', methods=['GET'])
def get_menu():
    """Return the full list of available menu items."""
    return jsonify({"menu": menu}), 200
```

# ASSIGNMENT 15

```python
# ====================================
# GET /order/{id} → Track order status
# ====================================
@app.route('/order/<int:order_id>', methods=['GET'])
def get_order(order_id):
    """Track the status of a specific order."""
    for order in orders:
        if order["id"] == order_id:
            return jsonify({"order": order}), 200
    return jsonify({"error": "Order not found"}), 404


# ====================================
# PUT /order/{id} → Update an existing order
# ====================================
@app.route('/order/<int:order_id>', methods=['PUT'])
def update_order(order_id):
    """Update items or status of an existing order."""
    data = request.get_json()
    for order in orders:
        if order["id"] == order_id:
            order.update({k: v for k, v in data.items() if k in order})
            return jsonify({"message": "Order updated successfully", "order": order}), 200
    return jsonify({"error": "Order not found"}), 404


# ====================================
# DELETE /order/{id} → Cancel an order
# ====================================
@app.route('/order/<int:order_id>', methods=['DELETE'])
def delete_order(order_id):
    """Cancel an existing order."""
    global orders
    for order in orders:
        if order["id"] == order_id:
            orders = [o for o in orders if o["id"] != order_id]
            return jsonify({"message": f"Order ID {order_id} cancelled successfully"}), 200
    return jsonify({"error": "Order not found"}), 404
```

```python
@app.errorhandler(405)
def method_not_allowed(e):
    return jsonify({"error": "Method not allowed"}), 405


@app.errorhandler(400)
def bad_request(e):
    return jsonify({"error": "Bad request"}), 400


# ========================================
# Run Flask Server
# ========================================
if __name__ == '__main__':
    app.run(debug=True)
```

# ASSIGNMENT 15

## EXPLANATION :

The `/menu` GET endpoint displays all available dishes with their names and prices.
The `/order` POST endpoint allows users to place new orders by sending a list of food items in JSON format.
Each order is automatically assigned a unique ID and stored with an initial status of "Order Placed."
The `/order/<id>` GET endpoint lets users check the status or details of a specific order.
Using the `/order/<id>` PUT endpoint, customers or staff can update an order's items or status.
The `/order/<id>` DELETE endpoint allows cancellation of an existing order by its ID.
All responses are structured as JSON with appropriate success or error messages.
Error handlers return clear messages for invalid input (400) and unsupported methods (405).
This API demonstrates the core CRUD operations—Create, Read, Update, and Delete—using Flask.
It can be easily extended to connect with a database for persistent storage or to handle authentication.