Unit 6: Web Services and APIs

6.1 Introduction to Web Services

Web services allow different applications to interact with each other regardless of their underlying platforms or programming languages. They facilitate communication between clients and servers using web technologies such as HTTP, XML, and JSON.

Example: Suppose an e-commerce website needs to check stock availability from a warehouse management system. A web service can expose stock data via APIs that the website queries.

Advantages

- **Interoperability:** Works across different platforms and programming languages.
- **Scalability:** Can handle multiple requests from different clients.
- **Standardized Communication:** Uses common protocols like HTTP, SOAP, and REST.

Disadvantages

- **Security Concerns:** Data transmitted over the internet can be intercepted.
- **Latency Issues:** Network delays can affect performance.
- **Complexity:** Requires proper implementation and maintenance.

6.2 Brief about Service-Oriented Architecture (SOA)

SOA is a design approach where software components provide services to other components via standardized interfaces. These services are loosely coupled, meaning they can operate independently while collaborating.

Example: A banking system could have separate services for authentication, transactions, and loan processing. Each service communicates through web services rather than being tightly integrated.

Key Characteristics of SOA:

1. **Interoperability** – Services use open standards like XML, JSON, and SOAP for communication.
2. **Reusability** – A service developed for one purpose can be reused in different applications.
3. **Standardized Communication –** Services exchange data using protocols like HTTP, SOAP, or REST.
4. **Autonomous Execution –** Each service functions independently with its own logic.

Disadvantages

- **Complex Implementation:** Requires careful design and management.
- **Performance Overhead:** Multiple service calls can slow down response times.
- **Security Challenges:** Requires robust authentication and authorization mechanisms.

How SOA Works?

SOA is typically structured around a set of **services**, each performing a specific task. Services communicate using **service contracts** (like WSDL for SOAP-based services), which define the format and rules for interaction.

Example: Think of an **online travel booking system** that consists of separate services:

- **Authentication Service**: Handles user login.
- **Hotel Service**: Provides hotel availability and booking.
- **Flight Service**: Manages flight reservations.
- **Payment Service**: Processes payments.

SOA Implementation Technologies

SOA can be implemented using:

- **SOAP-based Web Services** (Uses XML over HTTP, provides strict security)
- **RESTful Web Services** (Uses JSON/XML over HTTP, lightweight and flexible)
- **Microservices** (A more advanced version of SOA with fine-grained services)

Real-World Application of SOA

- **Banking Systems**: Separate services for transactions, customer support, loan processing, and fraud detection.
- **E-commerce Websites**: Independent services for payments, order processing, inventory management, and shipping.

- **Healthcare Systems**: Services for patient records, billing, appointments, and diagnostics.

Advantages of SOA

1. **Scalability** – Easily scale individual services as needed.
2. **Flexibility –** Modify and reuse services without disrupting the whole system.
3. **Efficiency –** Services are reusable across multiple applications, reducing development time.
4. **Better Integration** – SOA enables seamless interaction between different systems using industry standards.

6.3 SOAP (Simple Object Access Protocol)

SOAP is a protocol that enables web services to exchange structured data using XML over HTTP, SMTP, or other transport protocols. It ensures security, reliability, and extensibility.

Example: A payment gateway can use a SOAP-based web service to securely send transaction details, ensuring encryption and authentication. It's commonly used in **enterprise applications** (banking, insurance, telecom, etc.).

SOAP Message Structure

A typical SOAP message is an XML document structured like this:

```
xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!-- Optional: authentication, metadata, etc. -->
  </soap:Header>
  <soap:Body>
    <!-- Required: actual message data -->
  </soap:Body>
</soap:Envelope>
```

Example: Request to get the price of a stock

```
xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:stk="http://example.com/stock">
  <soapenv:Header/>
  <soapenv:Body>
    <stk:GetStockPrice>
      <stk:StockName>GOOG</stk:StockName>
    </stk:GetStockPrice>
  </soapenv:Body>
</soapenv:Envelope>
```

Advantages

- **High Security:** Supports WS-Security for secure transactions.
- **Standardized Protocol:** Works well in enterprise environments.
- **Supports Complex Operations:** Ideal for applications requiring strict data integrity.

Disadvantages

- **Heavy Payload:** Uses XML, which increases data size.
- **Slower Performance:** More processing overhead compared to REST.
- **Complex Implementation:** Requires additional configurations.

6.4 RESTful Web Services

Definition

REST (Representational State Transfer) is an architectural style for designing web services that use HTTP methods (GET, POST, PUT, DELETE) to interact with resources.

Example

A social media platform uses RESTful APIs to fetch user profiles, post updates, and retrieve comments.

Key Principles of REST

Principle	Description
Stateless	Each request from a client contains all the information needed — no session state is stored on the server.
Client-Server	Separation of concerns — the client handles the UI; the server handles data and logic.
Cacheable	Responses can be cached to improve performance.
Uniform Interface	Standardized way to interact with resources (using HTTP methods like GET, POST, etc.).
Resource-	Everything is a resource (user, order, product), accessed via URIs.

Principle	Description
Based	

HTTP Methods in REST

Method	Description	Example
GET	Retrieve data	/users/5
POST	Create new resource	/users
PUT	Update existing resource	/users/5
DELETE	Delete resource	/users/5
PATCH	Partially update resource	/users/5

RESTful API Example

Let's say we have a service that manages books.

Endpoint: /books

Action	HTTP Method	URI	Description
Get all books	GET	/books	List all books
Get one book	GET	/books/1	Get book with ID 1
Add new book	POST	/books	Create a new book
Update a book	PUT	/books/1	Update book with ID 1
Delete a book	DELETE	/books/1	Delete book with ID 1

Sample JSON Response (GET /books/1)

Json

```
{
  "id": 1,
  "title": "Clean Code",
  "author": "Robert C. Martin",
  "year": 2008
```

}

Advantages

1. Lightweight and fast
2. Human-readable (especially with JSON)
3. Easy to integrate with frontends
4. Widely supported (web, mobile, IoT)
5. Can be used over the open internet

Disadvantages

- **Less Security:** Does not have built-in security like SOAP.
- **Stateless:** Each request is independent, requiring additional authentication.
- **Limited Standards:** No strict rules for implementation.

6.6 RESTful APIs vs. SOAP

RESTful API vs SOAP API

Feature	RESTful API	SOAP API
Protocol	Uses HTTP/HTTPS	Uses its own protocol over HTTP, SMTP, etc.
Format	JSON, XML, HTML, plain text, etc.	Strictly XML
Message Structure	Flexible, no strict rules	Fixed structure: Envelope, Header, Body, Fault
Ease of Use	Simple, easy to learn	More complex and rigid
Performance	Lightweight, faster	Heavier due to XML and overhead
Security	SSL/TLS, OAuth, token-based auth	WS-Security (more comprehensive but complex)
Stateless	Yes	Can be stateless or stateful
Caching	Supported via HTTP caching	Not natively supported
Error Handling	HTTP status codes	<Fault> element in response
Best For	Web, mobile apps, public APIs	Enterprise-level, complex B2B communication

6.6 JSON and XML Data Formats

JSON vs XML: Overview

Feature	JSON (JavaScript Object Notation)	XML (eXtensible Markup Language)
Format Type	Lightweight data-interchange format	Markup language
Readability	Easy to read and write for humans	More verbose and harder to read
Data Size	Compact, smaller payloads	Verbose, larger payloads
Syntax	Key-value pairs using {} and []	Tag-based, like <tag>value</tag>
Data Types	Supports numbers, strings, booleans, arrays, objects	All values are strings (without schema)
Parsing Speed	Faster to parse	Slower due to complex structure
Comments	✕ Not supported (by default)	✓ Supported
Schema Support	JSON Schema (optional, lightweight)	XML Schema (XSD, very powerful)
Use Case	Web/mobile APIs, JavaScript-heavy apps	SOAP, configuration, legacy systems

JSON Example:

json

```
{
 "book": {
  "title": "Clean Code",
  "author": "Robert C. Martin",
  "year": 2008,
  "available": true
 }
}
```

XML Example:

```
xml
<book>
  <title>Clean Code</title>
  <author>Robert C. Martin</author>
  <year>2008</year>
  <available>true</available>
</book>
```

6.7 Building and Consuming APIs

Definition

Building an API involves creating endpoints that allow clients to interact with a server. Consuming an API means using an existing API to fetch or send data.

Example

A weather app consumes an API to get temperature data, while a developer builds an API to provide stock market updates.

Advantages

- **Efficiency:** Reduces development time by reusing APIs.
- **Scalability:** APIs can handle multiple clients.
- **Standardization:** Ensures consistent communication.

Disadvantages

- **Security Risks:** Requires proper authentication.
- **Dependency Issues:** Changes in an API can affect consumers.
- **Performance Overhead:** Poorly designed APIs can slow down applications.

6.7.1 Creating Simple RESTful APIs Using Flask

Steps to Create a Flask API

1. **Install Flask**

   ```
   pip install flask
   ```

2. **Create a Flask App**

   ```
   from flask import Flask, jsonify
   ```

```
app = Flask(__name__)

@app.route('/api/user', methods=['GET'])

def get_user():

    user = {"name": "Suraj", "age": 22, "skills": ["Python", "Flask", "Bootstrap"]}

    return jsonify(user)

    if __name__ == '__main__':

    app.run(debug=True)
```

3. **Run the API**

   ```
   python app.py
   ```

4. **Access the API** Open http://127.0.0.1:5000/api/user in a browser or use Postman.

Advantages

- **Easy to Implement:** Flask is beginner-friendly.
- **Lightweight:** Minimal dependencies.
- **Flexible:** Can be extended with additional features.

Disadvantages

- **Limited Built-in Features:** Requires extensions for advanced functionality.
- **Not Ideal for Large Applications:** Better suited for small projects.

6.4 RESTful Web Services

REST (Representational State Transfer) is an architectural style that uses HTTP methods like GET, POST, PUT, DELETE to perform operations on resources. RESTful services often use JSON or XML for data exchange.

Example: A weather forecasting service might offer a REST API where users send a GET request to retrieve temperature data.

Advantages

- **Lightweight:** Uses JSON, making it faster and efficient.
- **Scalability:** Easily handles large numbers of requests.

- **Simplicity:** Easier to implement compared to SOAP.

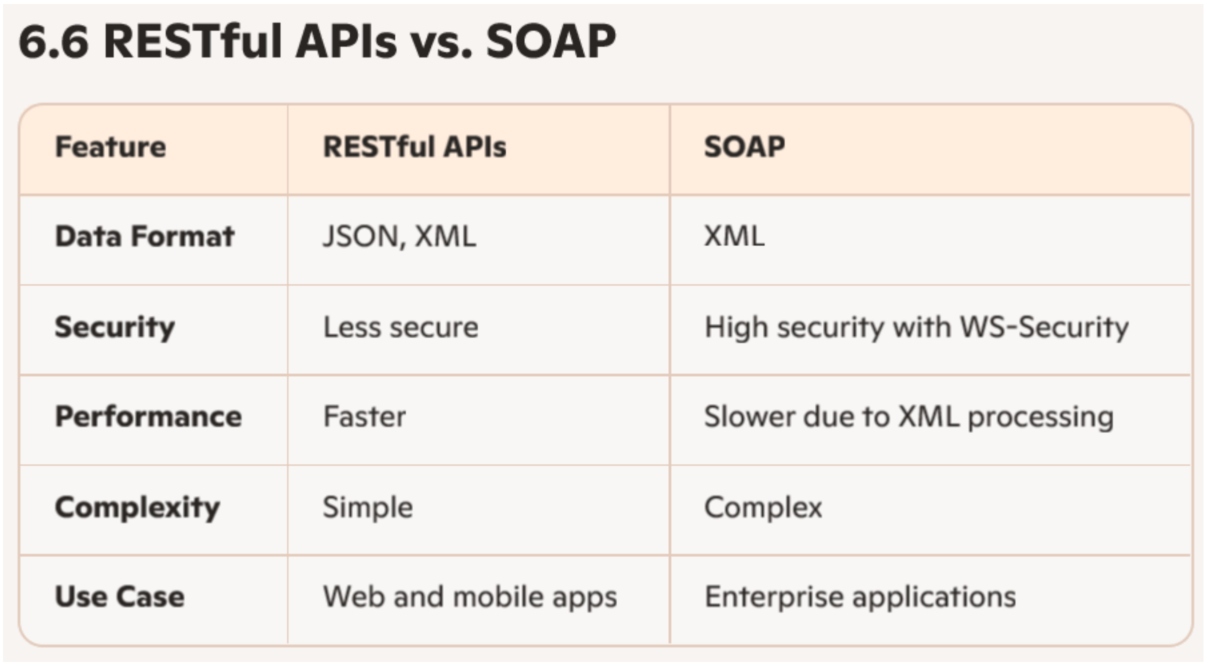
Disadvantages

- **Less Security:** Does not have built-in security like SOAP.
- **Stateless:** Each request is independent, requiring additional authentication.
- **Limited Standards:** No strict rules for implementation.

6.5 RESTful APIs vs. SOAP

- **SOAP**: Supports complex operations, provides built-in security, uses XML, requires a strict contract (WSDL).
- **REST**: Simpler, stateless, uses JSON/XML, flexible, widely adopted for web applications.

Example: Google Maps API is RESTful, whereas a secure financial system might use SOAP for transaction security.



6.6 RESTful APIs vs. SOAP

Feature	RESTful APIs	SOAP
Data Format	JSON, XML	XML
Security	Less secure	High security with WS-Security
Performance	Faster	Slower due to XML processing
Complexity	Simple	Complex
Use Case	Web and mobile apps	Enterprise applications

6.6 JSON and XML Data Formats

- **JSON (JavaScript Object Notation)**: Lightweight, human-readable, faster processing.
- **XML (Extensible Markup Language)**: Hierarchical, supports metadata, widely used in SOAP.
- Example: **JSON**

- json
- {
- "name": "Suraj",
- "age": 20,
- "city": "Lumbini"
- "skills": ["Python", "Flask", "Bootstrap"]
- }

- **XML**

```
<user>

  <name>Suraj</name>

  <age>22</age>

  <skills>

    <skill>Python</skill>

    <skill>Flask</skill>

    <skill>Bootstrap</skill>

  </skills>

</user>
```

6.7 Building and Consuming APIs

To build APIs, developers use frameworks such as Flask (Python) or Express (Node.js). Clients (web applications, mobile apps) consume these APIs through HTTP requests.

Example: A movie database API can expose a list of films, allowing a mobile app to retrieve movie details by sending requests.

Advantages

- **Efficiency:** Reduces development time by reusing APIs.
- **Scalability:** APIs can handle multiple clients.
- **Standardization:** Ensures consistent communication.

Disadvantages

- **Security Risks:** Requires proper authentication.
- **Dependency Issues:** Changes in an API can affect consumers.

- **Performance Overhead:** Poorly designed APIs can slow down applications.

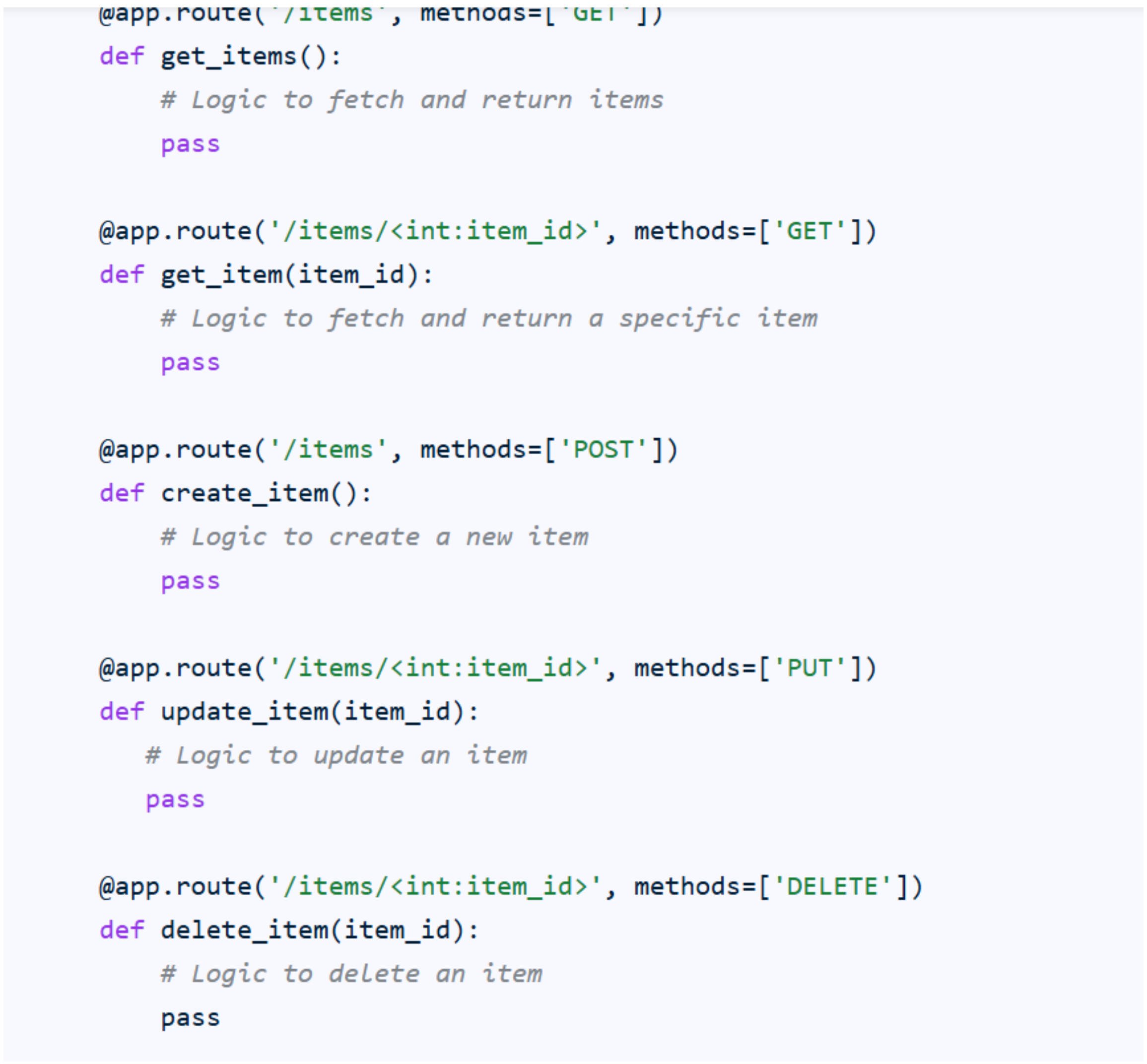
6.7.1 Creating Simple RESTful APIs Using Flask

1. Set up Flask:

- Install Flask using pip: pip install Flask.
- Create a Python file (e.g., app.py).
- Import Flask: from flask import Flask, jsonify, request.
- Initialize a Flask app: app = Flask(__name__).

2. Define Routes:

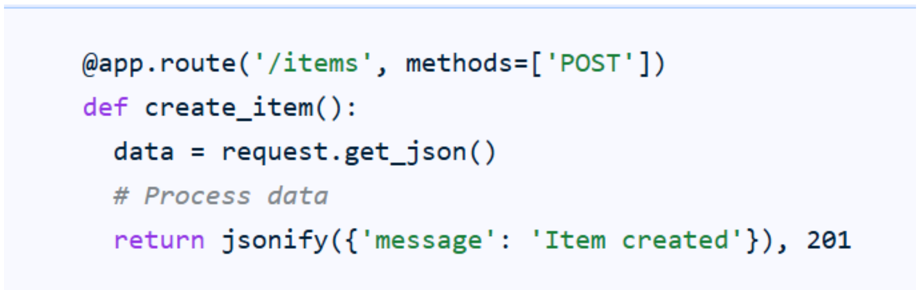
- Use the @app.route() decorator to define URL endpoints.
- Specify HTTP methods (e.g., GET, POST, PUT, DELETE).
- Example:

- 

```python
@app.route('/items', methods=['GET'])
def get_items():
    # Logic to fetch and return items
    pass

@app.route('/items/<int:item_id>', methods=['GET'])
def get_item(item_id):
    # Logic to fetch and return a specific item
    pass

@app.route('/items', methods=['POST'])
def create_item():
    # Logic to create a new item
    pass

@app.route('/items/<int:item_id>', methods=['PUT'])
def update_item(item_id):
   # Logic to update an item
   pass

@app.route('/items/<int:item_id>', methods=['DELETE'])
def delete_item(item_id):
    # Logic to delete an item
    pass
```

3. Handle Requests:

- Access request data using request.get_json() for JSON data, request.args for query parameters, and request.form for form data.
- Example:



```python
@app.route('/items', methods=['POST'])
def create_item():
  data = request.get_json()
  # Process data
  return jsonify({'message': 'Item created'}), 201
```

4. Return Responses:

- Use jsonify() to return JSON responses.
- Set appropriate HTTP status codes (e.g., 200 for success, 201 for created, 404 for not found).
- Example:

```
@app.route('/items/<int:item_id>', methods=['GET'])
def get_item(item_id):
    # Fetch item
    if item:
      return jsonify(item), 200
    return jsonify({'message': 'Item not found'}), 404
```

5. Run the App:

- Add if __name__ == '__main__': app.run(debug=True) to run the Flask development server.
- Execute the script: python app.py.

Key Concepts:

- **RESTful Principles:** Use HTTP methods correctly (GET for retrieval, POST for creation, PUT for updates, DELETE for deletion).
- **JSON:** Use JSON for data exchange.
- **Status Codes:** Provide appropriate HTTP status codes.
- **Flask:** Flask is a microframework, so it's lightweight and easy to use for small to medium-sized APIs.

Flask is a lightweight Python framework for building web services.

Example: Basic Flask API:

```
python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/hello', methods=['GET'])
def hello():
    return jsonify({"message": "Hello, Suraj!"})

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

Advantages

- **Easy to Implement:** Flask is beginner-friendly.
- **Lightweight:** Minimal dependencies.
- **Flexible:** Can be extended with additional features.

Disadvantages

- **Limited Built-in Features:** Requires extensions for advanced functionality.
- **Not Ideal for Large Applications:** Better suited for small projects.

Develop a REST API using Flask that performs CRUD operations on a student's table in the MySQL database.

1. Install the required packages

```
pip install Flask Flask-MySQLdb flask-restful
```

2. Create your MySQL database and table:

```
CREATE DATABASE school;

USE school;

CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    grade VARCHAR(10) NOT NULL
);
```

Flask App: app.py

```
from flask import Flask, request, jsonify
from flask_mysqldb import MySQL
from flask_restful import Resource, Api
```

```
app = Flask(__name__)
api = Api(app)

# MySQL Configuration
app.config['MYSQL_HOST'] = 'localhost'
app.config['MYSQL_USER'] = 'your_username'
app.config['MYSQL_PASSWORD'] = 'your_password'
app.config['MYSQL_DB'] = 'school'

mysql = MySQL(app)

# Helper function to convert DB rows to dictionaries
def student_to_dict(row):
    return {'id': row[0], 'name': row[1], 'age': row[2], 'grade': row[3]}

# CRUD Resources
class StudentList(Resource):
    def get(self):
        cur = mysql.connection.cursor()
        cur.execute("SELECT * FROM students")
        students = cur.fetchall()
        cur.close()
        return jsonify([student_to_dict(s) for s in students])

    def post(self):
        data = request.get_json()
        name = data['name']
        age = data['age']
        grade = data['grade']

        cur = mysql.connection.cursor()
        cur.execute("INSERT INTO students (name, age, grade) VALUES (%s, %s, %s)", (name,
age, grade))
        mysql.connection.commit()
        cur.close()
        return {'message': 'Student added successfully'}, 201

class Student(Resource):
    def get(self, student_id):
        cur = mysql.connection.cursor()
        cur.execute("SELECT * FROM students WHERE id = %s", (student_id,))
        student = cur.fetchone()
        cur.close()
        if student:
            return student_to_dict(student)
        return {'message': 'Student not found'}, 404
    def put(self, student_id):
        data = request.get_json()
        name = data['name']
        age = data['age']
        grade = data['grade']

        cur = mysql.connection.cursor()
        cur.execute("UPDATE students SET name=%s, age=%s, grade=%s WHERE id=%s",
(name, age, grade, student_id))
        mysql.connection.commit()
        cur.close()
        return {'message': 'Student updated successfully'}

    def delete(self, student_id):
        cur = mysql.connection.cursor()
        cur.execute("DELETE FROM students WHERE id = %s", (student_id,))
        mysql.connection.commit()
        cur.close()
        return {'message': 'Student deleted successfully'}

# API Routes
api.add_resource(StudentList, '/students')
api.add_resource(Student, '/students/<int:student_id>')

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

Sample API Endpoints

Method	**Endpoint**	**Description**
GET	/students	List all students
POST	/students	Add a new student
GET	/students/<id>	Get student by ID
PUT	/students/<id>	Update student by ID
DELETE	/students/<id>	Delete student by ID

Example JSON Body (POST/PUT)

```
{
  "name": "Alice",
  "age": 22,
  "grade": "A"
}
```