Universidad Politécnica de Yucatán
Data Engineering
Student: Carlo Alejandro Ek Palomo

9th quadrimester
Coding Practice: Research

REST APIs

1 Introduction

In modern software development, the ability to create robust and scalable web services has become increasingly important. REST APIs (Representational State Transfer Application Programming Interfaces) have emerged as the standard architectural style for building these services. This report explores the fundamentals of REST APIs, their implementation, and provides a detailed example using Python and Flask.

2 Understanding REST APIs

REST APIs are architectural guidelines for creating web services that focus on system resources. Developed by Roy Fielding in his 2000 doctoral dissertation, REST has become the predominant method for building web services due to its simplicity and effectiveness. A REST API uses HTTP methods to perform operations on resources, typically representing these resources as URLs and transferring data using JSON format.

The core principles of REST include statelessness, meaning each request contains all necessary information without relying on server-side sessions, and a uniform interface that simplifies the architecture. REST APIs communicate through standard HTTP methods: GET for retrieving resources, POST for creating new resources, PUT for updating existing resources, and DELETE for removing resources.

3 Implementation Example

To demonstrate these concepts, we have implemented a simple user management API using Python and Flask. This implementation showcases the fundamental CRUD (Create, Read, Update, Delete) operations that form the backbone of most REST APIs.

3.1 Basic Setup and Dependencies

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import uuid

app = Flask(__name__)
CORS(app)

users = {}
```

This initial section imports the necessary dependencies. Flask provides the web framework functionality, while flask-cors enables Cross-Origin Resource Sharing, allowing the API to accept requests from different domains. The UUID library generates unique identifiers for our users. We initialize our Flask application and create an empty dictionary to serve as our in-memory database.

3.2 Retrieving Users

```
@app.route('/api/users', methods=['GET'])
def get_users():
    return jsonify(list(users.values())), 200

@app.route('/api/users/<user_id>', methods=['GET'])
def get_user(user_id):
    user = users.get(user_id)
    if user:
        return jsonify(user), 200
    return jsonify({"error": "User not found"}), 404
```

These endpoints handle data retrieval. The first route returns all users, while the second retrieves a specific user by ID. The functions demonstrate REST's resource-oriented nature, with URLs clearly identifying the resources being accessed. The HTTP status codes 200 and 404 are used to indicate success or failure, following REST conventions.

3.3 Creating Users

```
@app.route('/api/users', methods=['POST'])
def create_user():
    data = request.get_json()

    if not data or 'name' not in data or 'email' not in data:
        return jsonify({"error": "Incomplete data"}), 400

    user_id = str(uuid.uuid4())
    new_user = {
        "id": user_id,
        "name": data['name'],
        "email": data['email'],
        "age": data.get('age'),
    }

    users[user_id] = new_user
    return jsonify(new_user), 201
```

The user creation endpoint demonstrates data validation and resource creation. It checks for required fields (name and email) and returns an appropriate error if they're missing. Upon successful creation, it returns HTTP status 201, specifically designated for resource creation, along with the newly created user's data.

3.4 Updating Users

```
@app.route('/api/users/<user_id>', methods=['PUT'])
def update_user(user_id):
    if user_id not in users:
        return jsonify({"error": "User not found"}), 404

    data = request.get_json()
    user = users[user_id]

    if 'name' in data:
        user['name'] = data['name']
    if 'email' in data:
        user['email'] = data['email']
    if 'age' in data:
```

```
        user['age'] = data['age']

    return jsonify(user), 200
```

The update endpoint showcases partial updates, allowing clients to modify specific fields without needing to send the entire user object. This flexibility is a key advantage of REST APIs, enabling efficient data transfer and reducing bandwidth usage.

3.5 Deleting Users

```
@app.route('/api/users/<user_id>', methods=['DELETE'])
def delete_user(user_id):
    if user_id not in users:
        return jsonify({"error": "User not found"}), 404

    del users[user_id]
    return jsonify({"message": "User successfully deleted"}), 200
```

The deletion endpoint demonstrates proper error handling and success confirmation. It returns appropriate status codes and messages, maintaining consistency in the API's response format.

4 Conclusion

This implementation demonstrates the core principles of REST APIs: resource-oriented design, stateless communication, and standard HTTP methods. The example shows how Python and Flask can be used to create a clean, maintainable API that follows REST conventions. While this implementation uses in-memory storage for simplicity, the principles demonstrated here scale well to more complex applications with persistent storage and additional features.

The consistent use of HTTP status codes, clear error messages, and intuitive URL structure makes the API both user-friendly and maintainable. This example serves as a foundation for understanding REST API development and can be extended to build more complex applications.