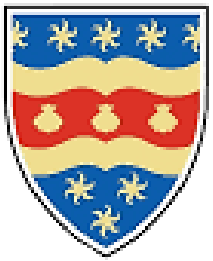


UNIVERSITY OF
PLYMOUTH

MAL2018 Information Management & Retrieval

Assessment 2

Student ID: BSCS2509261

Module Leader: Dr. Ang Jin Sheng

1. Introduction

This report documents the design and implementation of the **TrailService** micro-service, which I developed for the distributed Trail Application. The service acts as the backend core, enabling administrators to manage hiking trail data via a secure RESTful API. My main objective was to create a system that supports full Create, Read, Update, and Delete (CRUD) operations while strictly enforcing security through the University's authentication system.

The document outlines the entire development process, starting with the database schema design (3NF) and application structure, followed by a discussion on how I addressed LSEP issues like data privacy and security. It concludes with the technical implementation using **Python** and **Flask-RESTx**, and an evaluation of the system's functionality. The complete source code can be accessed via the link below:

GitHub Repository: https://github.com/farahazam27/TrailService_CW2

2. Background

The **Trail Application** is a distributed system designed to promote outdoor wellbeing by allowing users to create, explore, and rate hiking trails. The application is built using a micro-service architecture, where different features (like user profiles, badges, and comments) are handled by separate, independent services.

My specific role was to implement the **TrailService**. This micro-service acts as the core backend component responsible for managing trail data. It allows administrators to perform CRUD (Create, Read, Update, Delete) operations on trails. Crucially, it interfaces with the University's Authenticator API to ensure that only authorized users can modify the data, while still allowing the public to view trail details.

3. Design

3.1 Database Design (ERD)

I designed the database to follow **Third Normal Form (3NF)** rules. I separated the data into different tables (like Difficulty and RouteType) to avoid repetition. I also put the Coordinates in their own table so that one trail can have multiple location points without breaking the database rules.

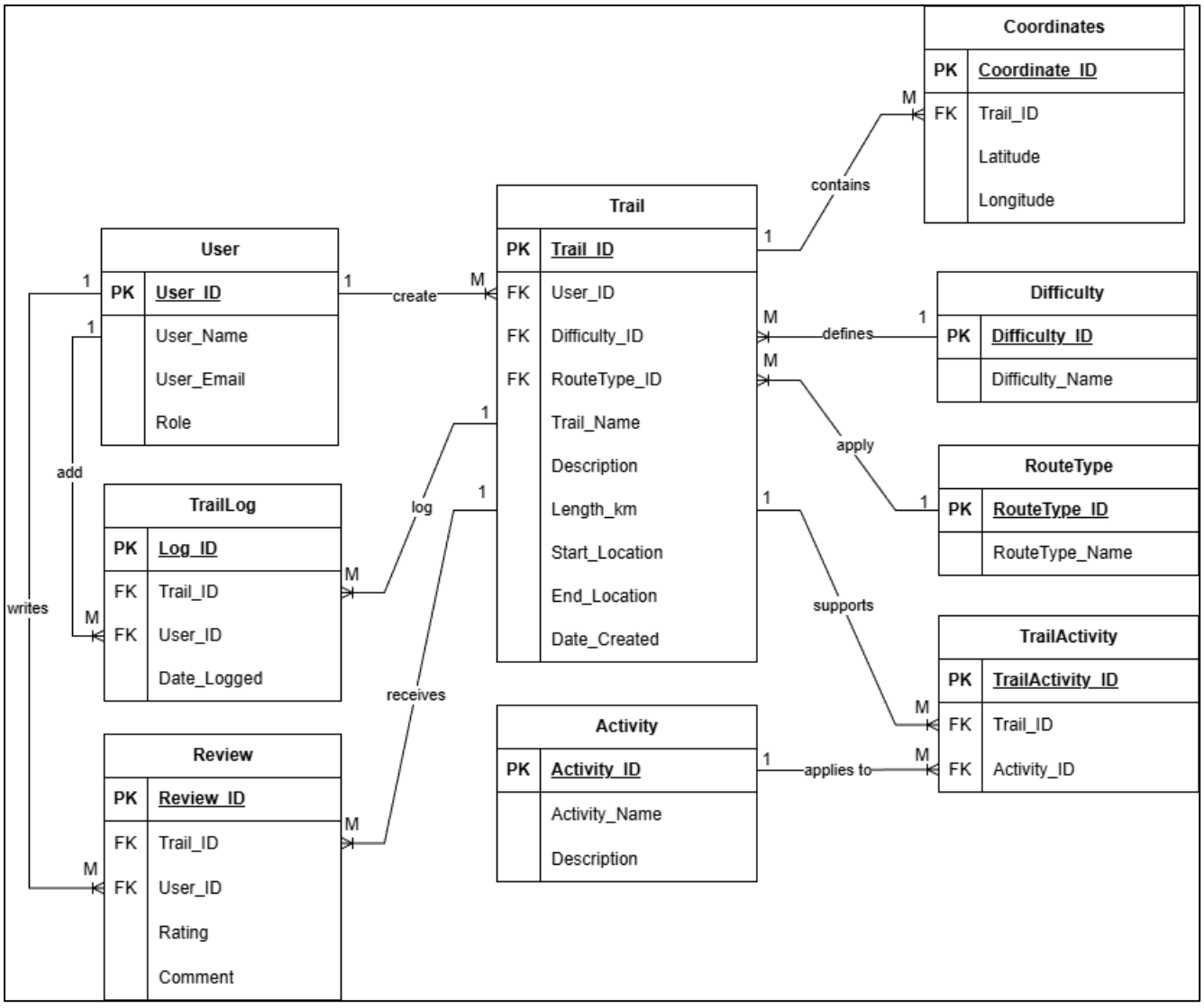


Figure 1: Logical Entity Relationship Diagram

As illustrated in Figure 1, the design separates the Trail entity from look-up tables such as Difficulty and RouteType. This normalization strategy ensures that descriptive attributes are consistent across all trails. Furthermore, the Coordinates are isolated in a separate table with a one-to-many relationship, allowing a single trail to consist of multiple location points without violating 1NF rules.

3.2 Application Design (UML Class Diagram)

I organized my Python code using the **Flask-RESTx** framework. As shown in Figure 2, I split the code into three main parts:

- **Controller Layer (TrailList, Trail):** These classes inherit from the Resource framework and handle HTTP requests (GET, POST, PUT, DELETE). They act as the entry point for the API.
- **Data Transfer Objects (TrailModel):** To ensure **Information Integrity** (an LSEP requirement), the system uses a strict DTO model. This model defines the expected data types (e.g., String, Float) and mandatory fields (Email, Password) before any data reaches the database.
- **Utility Layer (AppUtils):** Reusable logic, such as database connectivity and authentication verification, is encapsulated in helper functions to promote code maintainability.

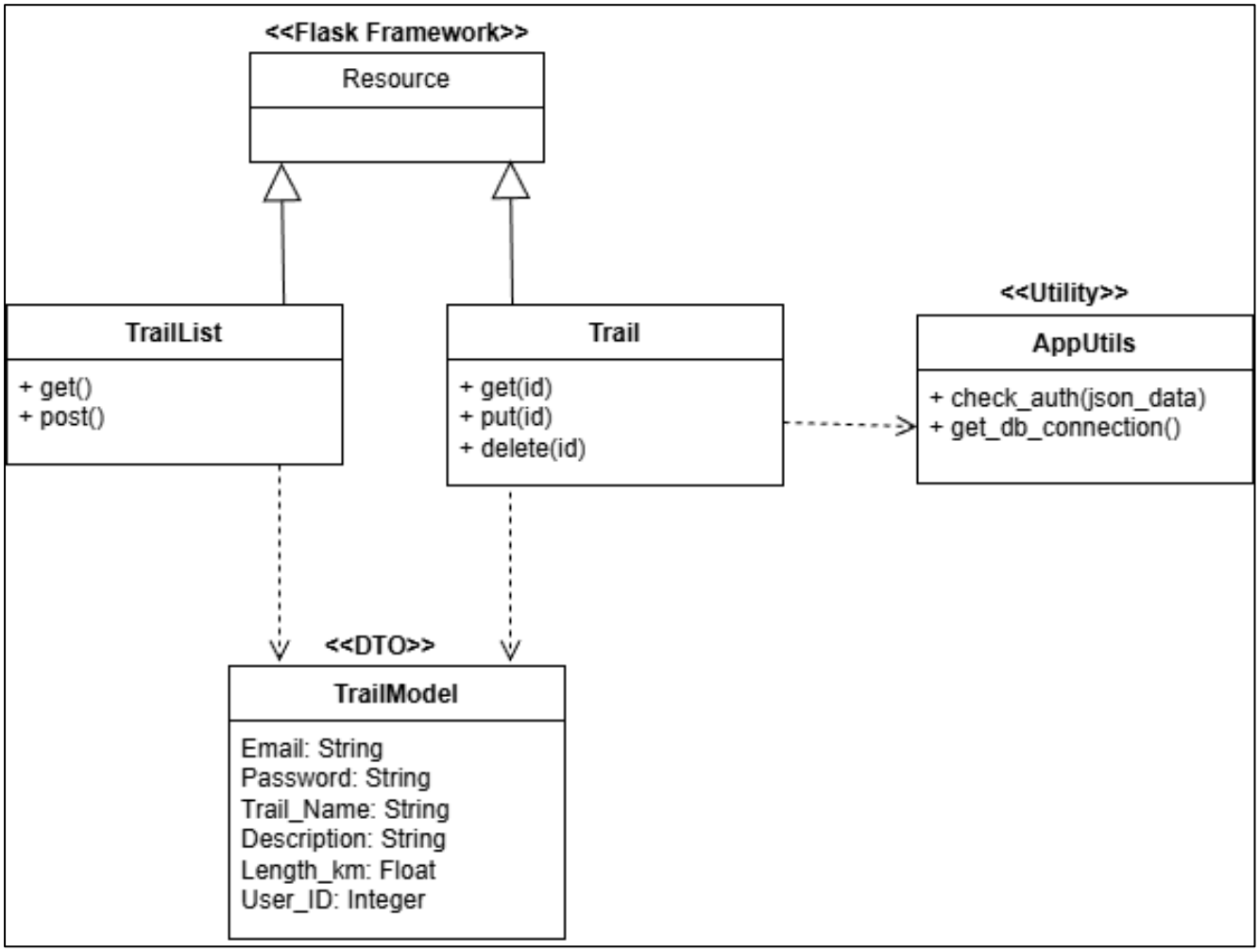


Figure 2: UML Class Diagram

3.3 Interaction Design (Sequence Diagram)

To show how the security works, I created a Sequence Diagram. This shows what happens when a user tries to create a new trail.

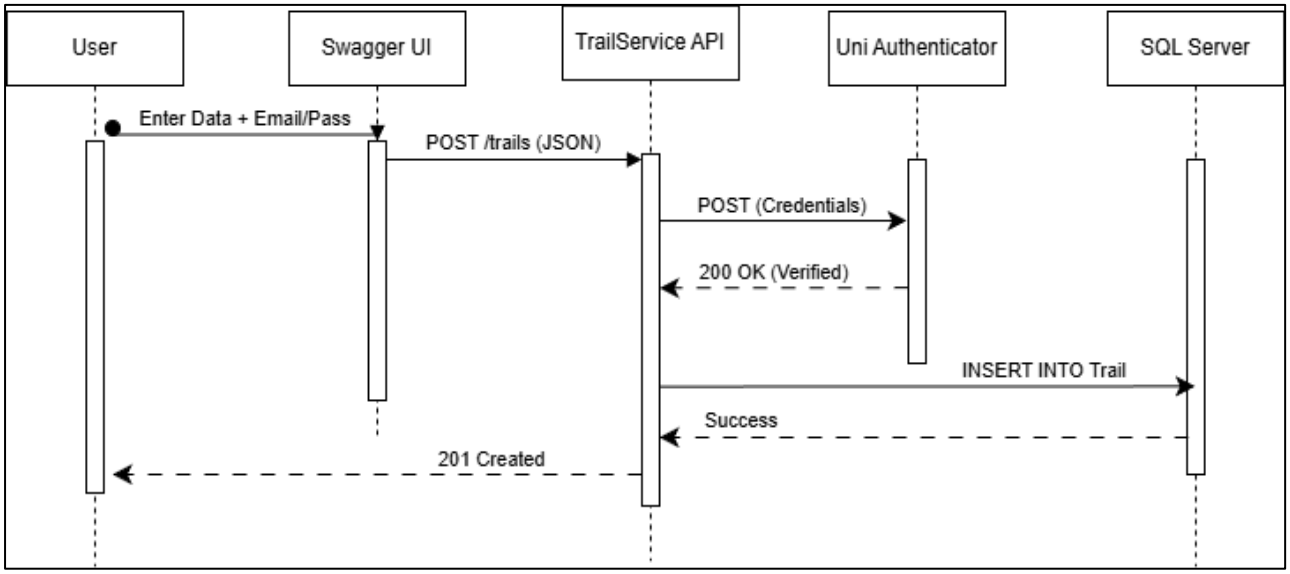


Figure 3: Secure Creation Sequence

Figure 3 shows the "Gatekeeper" system. When a user sends data to the API, my code stops and checks their Email and Password first. It sends these details to the University's Authenticator API. If the University says "True" (Verified), only then does my code allow the data to be saved into the SQL Database. This prevents unauthorized people from spamming the system.

4. Legal, Social, Ethical and Professional Issues (LSEP)

In designing this micro-service, I prioritized the protection of data and user trust. I addressed the core principles of LSEP by implementing specific technical controls for security, privacy, and integrity.

4.1 Information Security

A primary concern was preventing unauthorized modification of the trail data, which corresponds to the **OWASP Top 10** vulnerability regarding "Broken Access Control".

- **Authentication (The Gatekeeper):** As shown in the Sequence Diagram (Figure 3), I implemented a strict authentication check. Every time a user attempts to Add, Update, or Delete a trail, the system demands a valid University Email and Password. This ensures that only trusted users (such as 'Ada' or 'Tim') can alter the database, while the public is restricted to 'Read-Only' access.
- **SQL Injection Prevention:** To protect the database from malicious attacks, I used **Parameterized Queries** (using the ? placeholder in Python) for all SQL commands. This prevents attackers from injecting malicious code into the database via the input fields, ensuring the system remains secure.

4.2 Information Privacy

I designed the system to respect user privacy and comply with the **General Data Protection Regulation (GDPR)** principle of "Data Minimization" **(ICO, 2018)**.

- **Data Minimization:** My Trail database only stores the User_ID (a number). I do **not** store the user's password, name, or home address in my trail table.
- **Separation:** All personal user details are handled by the University's external Authenticator API, not my database. This means that even if the TrailService database were compromised, no user credentials or passwords would be exposed.

4.3 Information Integrity

I made sure the data stays accurate and remains consistent and accurate.

- **Foreign Keys:** I implemented Foreign Keys for attributes such as Difficulty_ID and RouteType_ID. This enforcement means it is impossible to enter invalid data (e.g., typing "Harddd" instead of selecting the ID for "Hard"). The database will automatically reject any data that does not match the pre-defined categories.
- **Data Validation:** My Python code checks the data before sending it to the database. For example, it converts the length numbers to the right format so the database doesn't crash.

5. Implementation

I built the micro-service using **Python** and the **Flask-RESTx** framework. I chose this setup because it is lightweight and allows me to create a professional API with documentation quickly.

5.1 Key Technologies

- **Flask-RESTx:** This library helped me build the API endpoints and automatically created the blue Swagger UI page.
- **PyODBC:** I used this tool to connect my Python code to the SQL Server database.
- **Requests:** I used this library to talk to the external University Authenticator API.
- **GitHub:** I used GitHub to manage my code versions (app.py and requirements.txt).

5.2 Handling Authentication (Security)

For the "Create Trail" feature, I wrote a function called check_auth. This function takes the user's email and password from the request and sends it to the University API.

```python
def get_db_connection():
    try:
        conn = pyodbc.connect(conn_str)
        return conn
    except Exception as e:
        print(f"Database connection failed: {e}")
        return None

def check_auth(json_data):
    """Reusable function to check University Auth"""
    user_email = json_data.get('Email')
    user_password = json_data.get('Password')

    if not user_email or not user_password:
        return False, "Email and Password are required."

    auth_url = "https://web.socem.plymouth.ac.uk/COMP2001/auth/api/users"
    credentials = {"email": user_email, "password": user_password}

    try:
        response = requests.post(auth_url, json=credentials)
        if response.status_code != 200:
            return False, "Authentication Failed! Invalid credentials."

        verified_user = response.json()
        if verified_user[1] != 'True':
            return False, "Authentication Failed!"

        return True, "Success"
    except Exception as e:
        return False, f"Service Error: {str(e)}"
```

Figure 4: Authentication Code

This code checks if the response from the University is "True". If it is, the program continues. If not, it stops and sends a "401 Unauthorized" error.

5.3 Database Connection & Data Fixing

I used a secure connection string to link to the database. During development, I faced a bug where Decimal numbers from the database (like trail length) caused an error. I fixed this by writing code to convert them into float in Python before sending the JSON response.

```python
def get_db_connection():
    try:
        conn = pyodbc.connect(conn_str)
        return conn
    except Exception as e:
        print(f"Database connection failed: {e}")
        return None
```

Figure 5: Database Connection Code

```python
for row in cursor.fetchall():
    row_dict = dict(zip(columns, row))

    for key, value in row_dict.items():
        if isinstance(value, Decimal):
            row_dict[key] = float(value)
    trails.append(row_dict)
```

Figure 6: Logic to convert Decimal types to Float for JSON compatibility

5.4 API Endpoints & Routing

I structured the API using **RESTful principles**, ensuring logical resource management. I defined two main routes using the Flask-RESTx @ns.route decorator:

Collection Route (/trails/):

- **GET:** Retrieves a JSON list of all available trails.
- **POST:** Allows authorized users to create a new trail.

Single Resource Route (/trails/<int:id>):

- **GET:** Retrieves details for a specific trail ID.
- **PUT:** Updates a specific trail (Authorized).
- **DELETE:** Removes a specific trail (Authorized).

```python
@ns.route('/')
class TrailList(Resource):
    @ns.doc('list_trails')
    def get(self):
        """List all trails (Public - No Auth needed)"""
        conn = get_db_connection()
        if not conn:
            return {"error": "Database connection failed"}, 500

        cursor = conn.cursor()
        try:
            cursor.execute("SELECT * FROM CW2.TrailDetailsView")
            trails = []
            columns = [column[0] for column in cursor.description]
            for row in cursor.fetchall():
                row_dict = dict(zip(columns, row))

                for key, value in row_dict.items():
                    if isinstance(value, Decimal):
                        row_dict[key] = float(value)
                trails.append(row_dict)
            return trails, 200
        except Exception as e:
            return {"error": str(e)}, 500
        finally:
            conn.close()
```

Figure 7: API Route Definitions in Python

This implementation directly reflects the structure defined in the **UML Class Diagram (Figure 2)**, where the TrailList and Trail Python classes correspond exactly to the controllers designed in the modeling phase

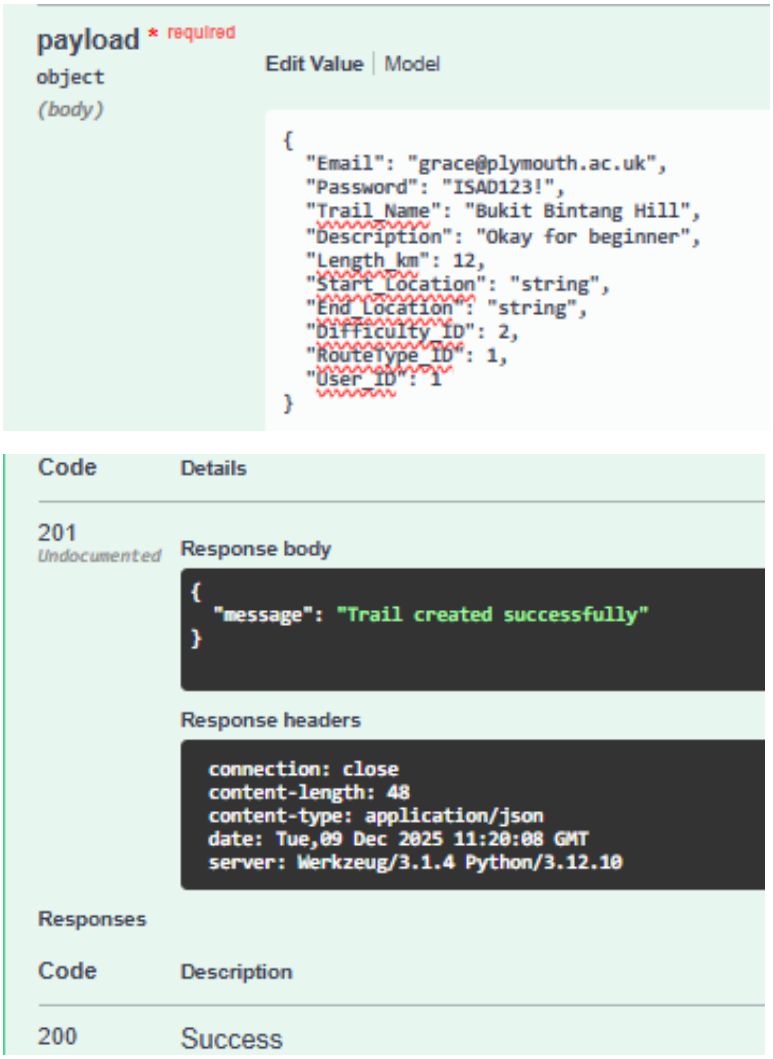
6. Evaluation

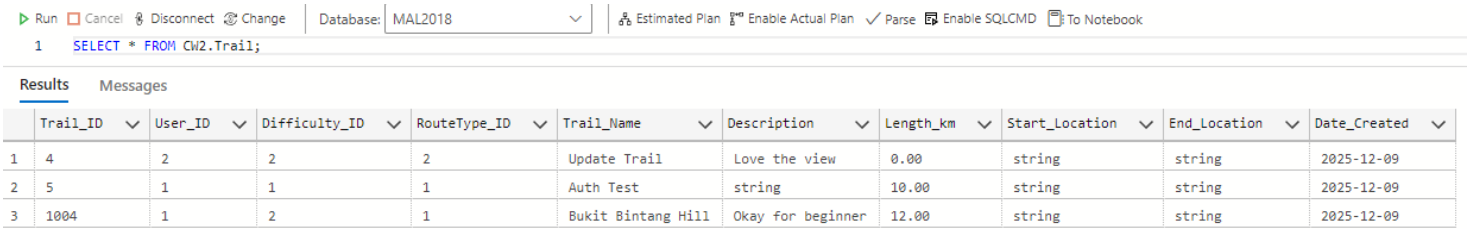
To make sure the API works correctly, I tested every feature manually using the **Swagger UI** webpage.

6.1 Functional Testing (CRUD)

I verified that I can Create, Read, Update, and Delete trails.

Test 1: Create (POST) I entered valid trail data and a correct University password. The system successfully added the trail (Code 201).





	Trail_ID	User_ID	Difficulty_ID	RouteType_ID	Trail_Name	Description	Length_km	Start_Location	End_Location	Date_Created
1	4	2	2	2	Update Trail	Love the view	0.00	string	string	2025-12-09
2	5	1	1	1	Auth Test	string	10.00	string	string	2025-12-09
3	1004	1	2	1	Bukit Bintang Hill	Okay for beginner	12.00	string	string	2025-12-09

Figure 8: Successful creation of a new trail

Test 2: Read Trails (GET) I checked the list of trails to make sure the new data was actually saved in the database.

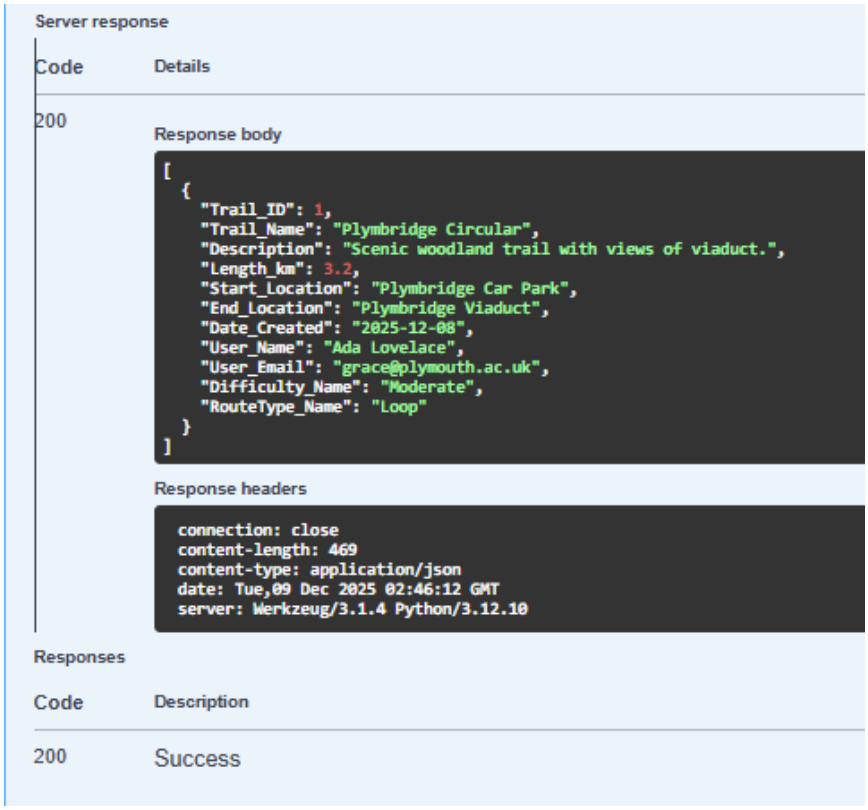


Figure 9: Retrieving the list of trails

Test 3: Update a Trail (PUT) I tested the update feature by changing a trail's name. The system updated the record without errors.

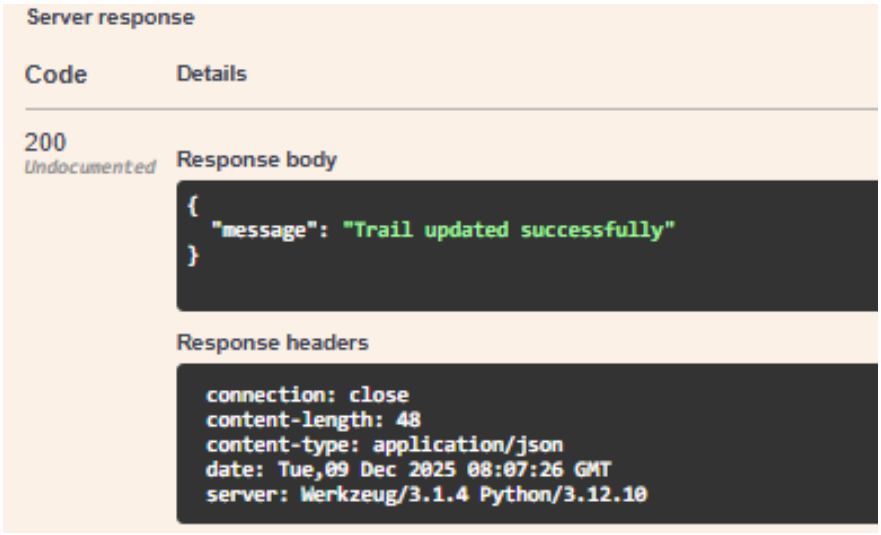


Figure 10: Updating an existing trail record

Test 4: Delete a Trail (DELETE) I deleted a trail using its ID. The system returned "204 No Content", which means it was deleted.

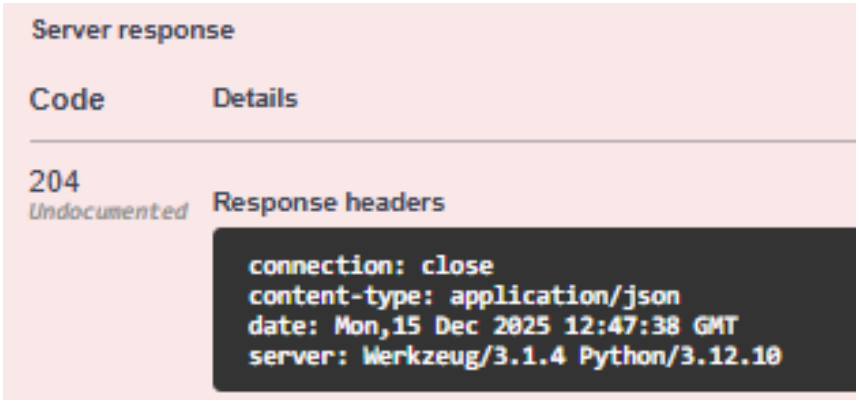


Figure 11: Successful deletion of a trail

6.2 Security Testing (LSEP)

I purposefully tried to create a trail using a **wrong password** to see if the system would block me. As expected, the API rejected the request.

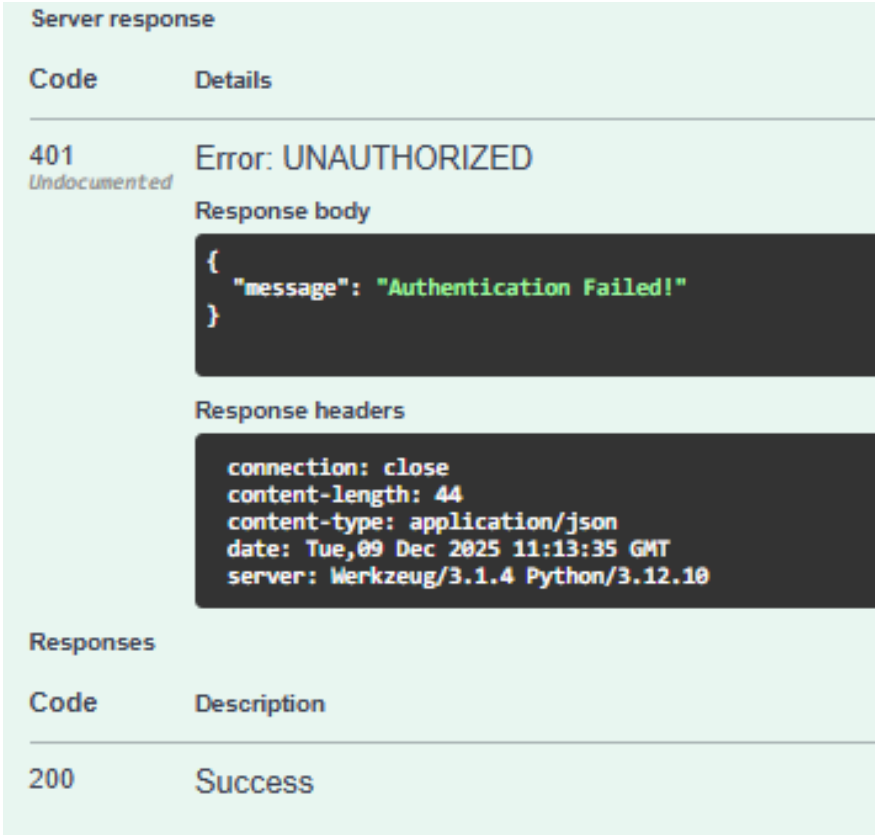


Figure 12: Security test blocked by the system

6.3 Reflection & Future Work

Overall, the API works well and successfully meets the assessment requirements.

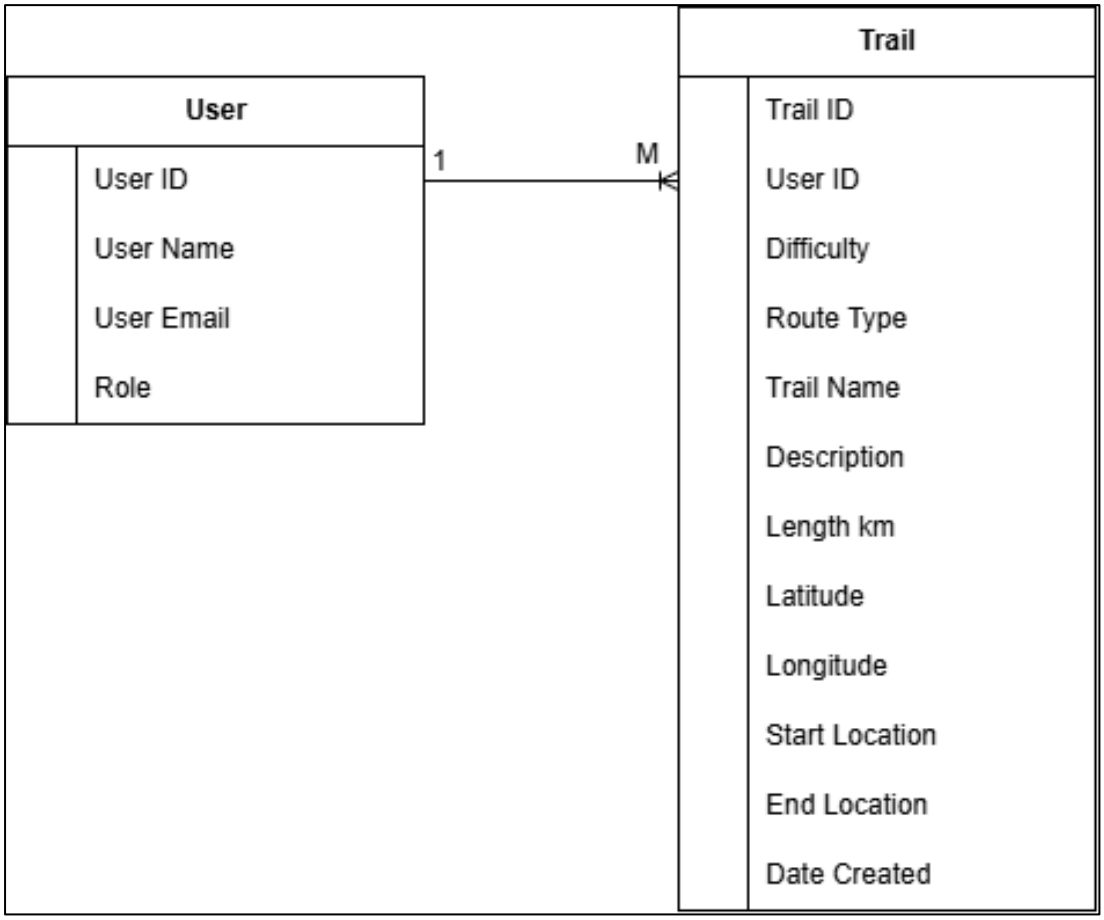
- **Strength (Authentication):** The strongest feature is the **Dynamic Authentication**. Instead of hardcoding a single user, my system connects to the University API, allowing any valid user (like Ada or Tim) to log in with their own credentials.
- **Weakness (User Interface):** A current limitation is that the API does not have a "Frontend" website. This means users must rely on the technical Swagger UI to interact with the system, which is not user-friendly for non-technical people.
- **Future Improvements:** In the future, I would like to build a simple HTML web page for hikers. I also plan to integrate the **Google Maps API** to visualize the trail coordinates on an actual map rather than just listing them as numbers.

7. References

1. **Flask-RESTx** (2025) *Flask-RESTx Documentation*. Available at: https://flask-restx.readthedocs.io/ (Accessed: 13 December 2025).
2. **ICO** (2018) *Guide to the General Data Protection Regulation (GDPR)*. Information Commissioner's Office. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/ (Accessed: 13 December 2025).
3. **Microsoft** (2025) *SQL Server Technical Documentation*. Available at: https://learn.microsoft.com/en-us/sql/ (Accessed: 13 December 2025).
4. **OWASP** (2021) *OWASP Top 10:2021*. The Open Web Application Security Project. Available at: https://owasp.org/Top10/ (Accessed: 13 December 2025).

APPENDIX

Appendix A: Initial Entity Relationship Diagram (ERD)



Note: This initial design was refined during the development process. The tables were normalized to Third Normal Form (3NF) to produce the final logical design presented in Section 3.1.