MAL2018

Information Management & Retrieval

Assessment 2

PREPARED BY:

Student ID: BSSE2506008

Course: BSC(HONS) Computer Science (Software Engineering)

Lecturer: DR. ANG JIN SHENG

Table of Contents

1.0 Introduction

This report documents the design and implementation of the TrailService microservice, a key component of a wellbeing trail discovery application. The TrailService provides comprehensive CRUD (Create, Read, Update, Delete) operations for managing trails, supporting an engaging platform for outdoor exploration. Moreover, the microservice integrates with an Authenticator API to handle user authentication. All application data is stored in a normalized Microsoft SQL Server database to ensure consistency, integrity, and efficient querying.

The report is structured into six main sections.The Background introduces the application context and the role of the microservice. The Design section presents the logical Entity Relationship Diagram (ERD), UML diagram, and microservice architecture. The Legal, Social, Ethical, and Professional (LSEP) Considerations section discusses how considerations of information privacy, security, integrity, and data preservation were addressed throughout the design process. The Implementation section provides details of the server-side code and database deployment. Finally, the Evaluation reviews the testing process, assesses system performance and identifies areas for future enhancement.

- **GitHub repository:** https://github.com/neohjiayi04/MAL2017_CW2.git
- **Hosted microservice:** http://127.0.0.1:5000/ui/

2.0 Background

TrailService is a RESTful microservice responsible for managing wellbeing trail data within a microservice-oriented architecture. It provides structured access to trail information stored in a Microsoft SQL Server database, including metadata such as trail name, location, difficulty level, route type, visibility, and estimated completion time. The service also supports full CRUD operations for trails while delegating authentication to the external COMP2001 Authenticator API to ensure secure credential handling and a clear separation of concerns.

The microservice is implemented in Python using Flask and Connexion to enable OpenAPI-driven development and automatic Swagger documentation. SQLAlchemy and Marshmallow provide secure database interaction and request validation, enhancing the system's reliability, maintainability, and overall security posture.

3.0 Design

3.1 Microservice Architecture

The TrailService is implemented as an independent RESTful microservice using a layered architecture that separates API handling, business logic, and data access. This design enforces clear responsibility boundaries and supports maintainability and independent evolution of the service.

At a system level, client applications communicate with the TrailService through a REST API, while persistent data is managed using a Microsoft SQL Server database. User authentication is externalised to the COMP2001 Authenticator API, ensuring that credential validation is centralised and that no sensitive authentication data is stored within the microservice.

At an implementation level, incoming HTTP/JSON requests are handled by the Flask and Connexion API layer, where they are validated against the OpenAPI specification defined in swagger.yml. Validated requests are then processed by the business logic layer, with all database interactions handled by the data access layer. This interaction between layers is illustrated in Figure 1.

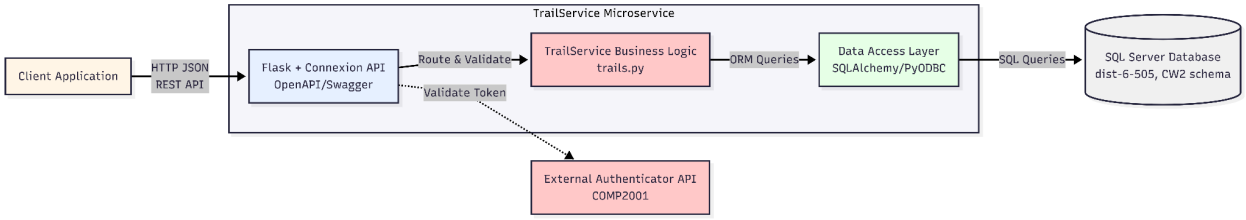


Figure 1: TrailService Architecture

3.2 Database Design

The database schema evolved from the normalized Third Normal Form (3NF) relations developed in CW1 and was implemented in the CW2 schema on Microsoft SQL Server. The ERD (Figure 2) illustrates the core entities, attributes, and relationships that support the TrailService microservice. The design minimizes data redundancy, enforces referential integrity through foreign key constraints, and supports future extensibility. The purpose of each entity is summarised in Table 1.

Entity	Purpose
Users	Stores user information and roles.
Trail	Core entity representing a hiking trail, including metadata such as name, visibility, route, difficulty, and creator.
Location	Stores geographic information including region, country, and location name.
Trail_Point	Stores ordered GPS coordinates for detailed trail mapping.
Route	Defines the trail structure (Loop, Out-and-Back, Point-to-Point).
Difficulty	Defines trail difficulty levels (Easy, Moderate, Hard, Strenuous).
Feature	Stores trail features such as waterfall, city walks, and lakes.
Trail_Feature	A junction table implementing the many-to-many relationship between *Trail* and *Feature*.
Trail_Log	An audit table populated automatically via SQL triggers whenever a new trail is created.

Table 1: Database Entities and Their Purpose

Although the database schema contains multiple supporting entities, the external REST API only exposes CRUD operations for *Trail*. The remaining entities act as internal reference and helper tables, ensuring metadata consistency, referential integrity, and audit logging.

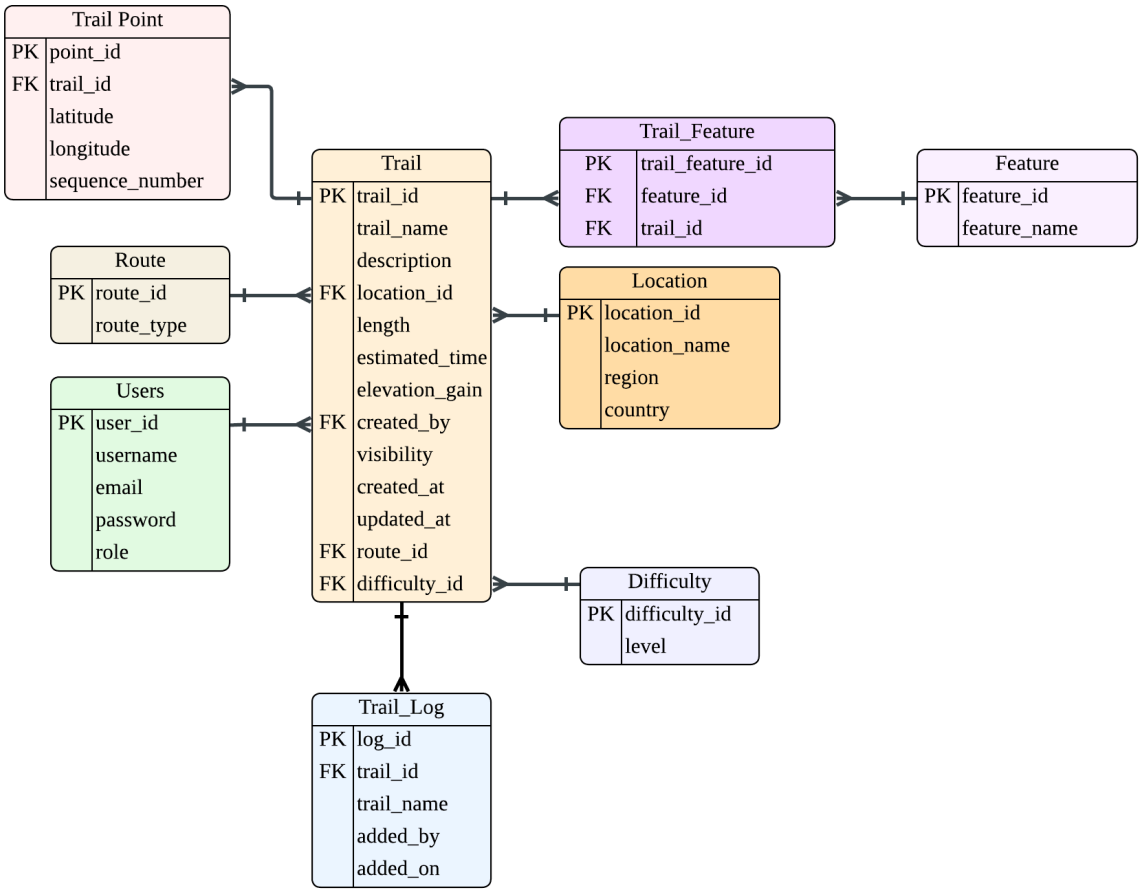


Figure 2: ERD for Microservice

3.3 UML Class Diagram

The UML class diagram (Figure 3) models the structural relationships between the core domain entities within the TrailService, demonstrating how the logical data model maps to the service's object structure while maintaining clear responsibility boundaries.

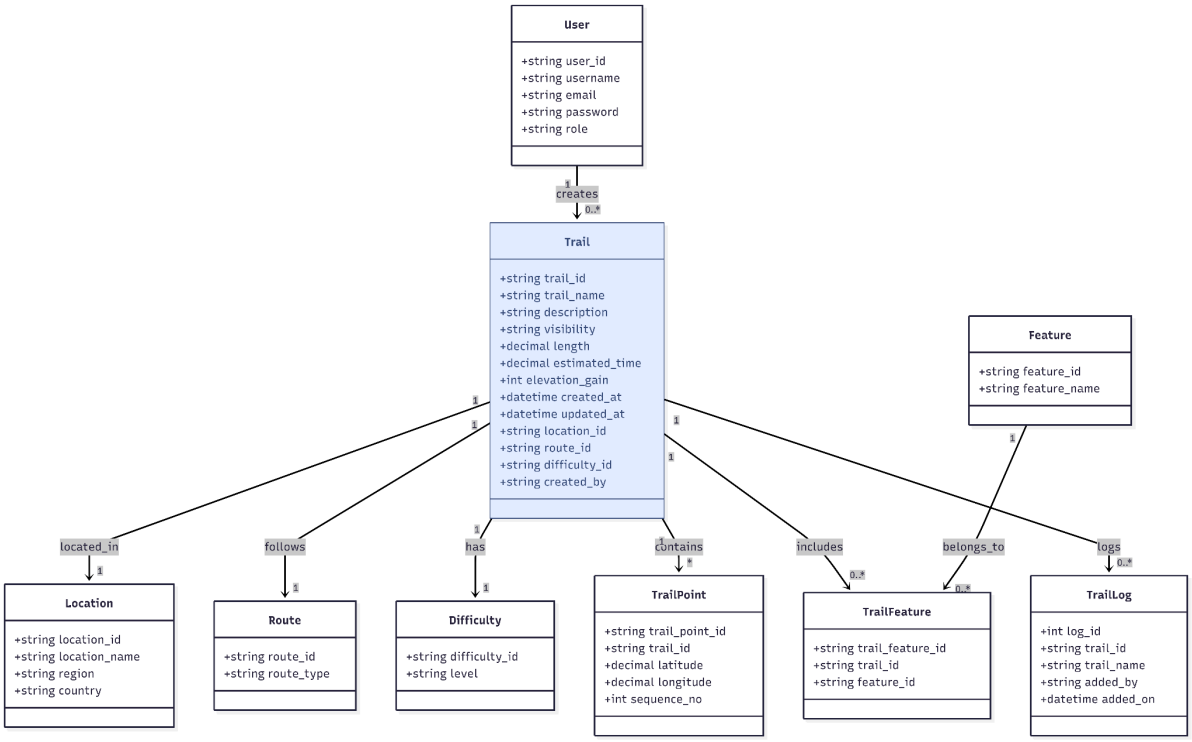
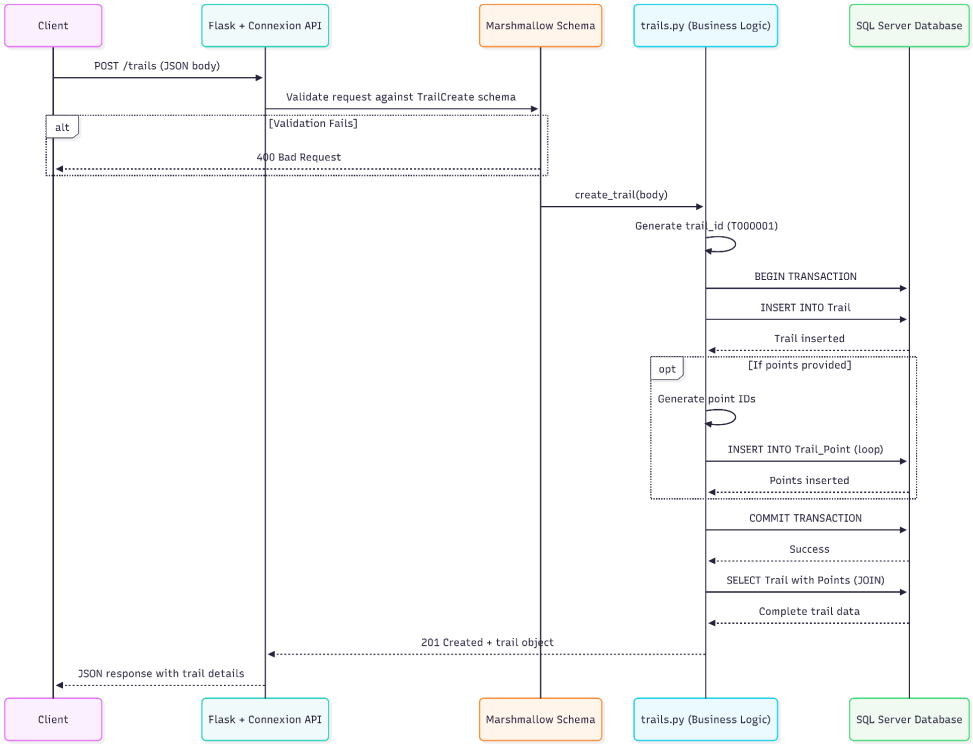


Figure 3: UML Class Diagram

3.4 API Interaction



The sequence diagram shows the complete workflow for creating a trail via POST /trails. After receiving trail data and optional GPS waypoints, the Flask–Connexion API validates the request using the TrailCreate Marshmallow schema. Once validated, the service generates sequential IDs for the trail and its waypoints, then performs all inserts within a single transaction to ensure atomicity and rollback on failure. The newly created trail is then retrieved, serialized, and returned to the client with a 201 Created response.

4.0 Legal, Social, Ethical and Professional (LSEP) Considerations

Legal Compliance

The system follows key EU General Data Protection Regulation (GDPR) principles, including data minimisation and secure processing as defined in Articles 5 and 32 (European Parliament, 2016). Only essential trail metadata and user identifiers are stored, while authentication is delegated to the external COMP2001 Authenticator API. Users can exercise GDPR rights through REST endpoints that support data access and deletion.

Information Security

Security is enforced through layered controls. SQLAlchemy parameterised queries mitigate SQL injection attacks (Sling Academy, 2024), and Marshmallow validates all incoming data. All communication uses HTTPS/TLS to ensure confidentiality in transit (OWASP, 2021). Authentication is delegated to the external COMP2001 Authenticator API, ensuring credentials are never stored or processed by the TrailService itself. This reduces credential exposure and supports secure separation of concerns. Error messages are intentionally generic to avoid exposing internal system details (Pfleeger & Pfleeger, 2015).

Data Integrity

Data integrity is ensured through a combination of database constraints and application-level validation. Primary and foreign key constraints enforce referential integrity, while SQLAlchemy type definitions and Marshmallow validation rules prevent invalid data from being persisted. An audit log (*Trail_Log*) records trail creation events, supporting accountability and traceability in line with responsible data governance practices.

Data Preservation

SQL Server's transaction logging and backup mechanisms provide durability and protection against data loss (Microsoft, 2023). Timestamp fields support chronological tracking, while referential constraints and audit logs preserve historical context, aligning with long-term digital preservation principles (Digital Preservation Coalition, 2015).

Social Impact

The TrailService microservice contributes positively to society by promoting outdoor activity, physical and mental wellbeing, and environmental awareness through community-shared trails. This demonstrates a responsible use of technology that supports public health and social engagement.

Ethical and Professional Standards

The system aligns with the ACM Code of Ethics and the BCS Code of Conduct by prioritising privacy, harm prevention, and responsible data handling (ACM, 2018; BCS, 2022). Endpoints expose only necessary information, enforce authorisation, and provide clear documentation to support inclusive and accessible development.

5.0 Implementation

5.1 Database Implementation

The database schema designed in Section 3.2 was implemented in Microsoft SQL Server under the CW2 schema (Figure 4). All entities, relationships, and constraints defined in the ERD were created, including primary and foreign keys to enforce referential integrity. Sample data was inserted into all tables to support testing and demonstration (Figure 5).

SQLAlchemy ORM models (Figure 6) map directly to the database schema, ensuring consistency between the database and application layers while preventing SQL injection through parameterized queries.

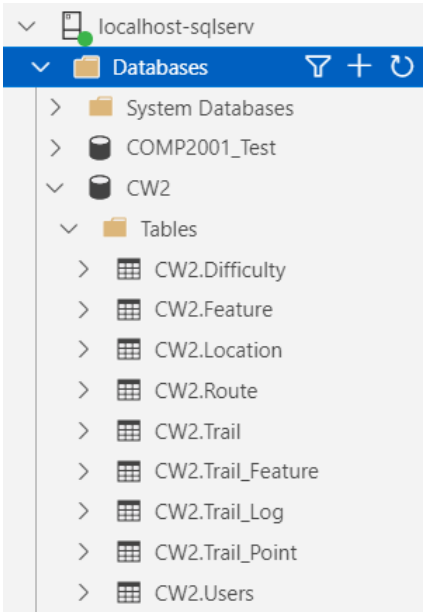


Figure 4: Database tables under CW2 schema

	trail_id	trail_name	description	visibility	created_at	updated_at
1	T000001	Bovisand to Jennycliff	Coastal walk in South Devon AONB along the coastal path g…	full	2025-11-18 11:01:00.000	2025-11-18 11:01:00.000
2	T000002	Sri Bintang Hill	Bukit Sri Bintang (Bukit Pelangi) is a popular hiking des…	full	2025-11-18 11:02:00.000	2025-11-18 11:02:00.000
3	T000003	Plymbridge Circular	Gentle circular walk through ancient oak woodlands beside…	full	2025-11-18 11:03:00.000	2025-11-18 11:03:00.000
4	T000004	Kingsley Hill Peak Loop	The loop trail consists of a dirt trail with several stee…	full	2025-11-18 11:03:00.000	2025-11-25 18:03:00.000

location_id	length	estimated_time	elevation_gain	route_id	difficulty_id	created_by
L000002	5.80	2.00	161	R000002	D000002	U000001
L000005	2.30	1.50	198	R000001	D000002	U000001
L000001	5.00	2.00	147	R000001	D000001	U000001
L000006	3.40	1.50	185	R000001	D000002	U000001

Figure 5: Sample Data in *Trails* table

```python
class Trail(db.Model):
    __tablename__ = 'Trail'
    __table_args__ = {'schema': 'CW2'}

    trail_id = db.Column(db.String(7), primary_key=True)
    trail_name = db.Column(db.String(100), nullable=False)
    description = db.Column(db.Text, nullable=True)
    visibility = db.Column(db.String(20), nullable=False)
    created_at = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, nullable=False, default=datetime.utcnow, onupdate=datetime.utcnow)
    location_id = db.Column(db.String(7), db.ForeignKey('CW2.Location.location_id'), nullable=True)

    length = db.Column(Numeric(5, 2, asdecimal=True), nullable=True)
    estimated_time = db.Column(Numeric(4, 2, asdecimal=True), nullable=True)

    elevation_gain = db.Column(db.Integer, nullable=True)
    route_id = db.Column(db.String(7), db.ForeignKey('CW2.Route.route_id'), nullable=False)
    difficulty_id = db.Column(db.String(7), db.ForeignKey('CW2.Difficulty.difficulty_id'), nullable=False)
    created_by = db.Column(db.String(7), db.ForeignKey('CW2.Users.user_id'), nullable=True)
```

Figure 6: SQLAlchemy ORM Model

5.2 API Implementation

The REST API is defined using an OpenAPI 3.0 specification (swagger.yml). Connexion automatically generates endpoints, validates requests, and routes them to the appropriate handler functions before business logic is executed (Figure 7).

CRUD operations are fully implemented for Trail resources (Table 2). GET requests allow public retrieval of trail data, while POST, PUT, and DELETE requests allow making changes.

Action	HTTP Verb	URL Path	Description
Read trail	GET	/trails /trails/{id} /trails/detailed	Retrieve all trails Retrieve specific trail Retrieve all trails with details
Create trail	POST	/trails	Create new trail
Update trail	PUT	/trails/{id}	Update existing trail
Delete trail	DELETE	/trails/{id}	Remove trail

Table 2: Trail RESTful Endpoints

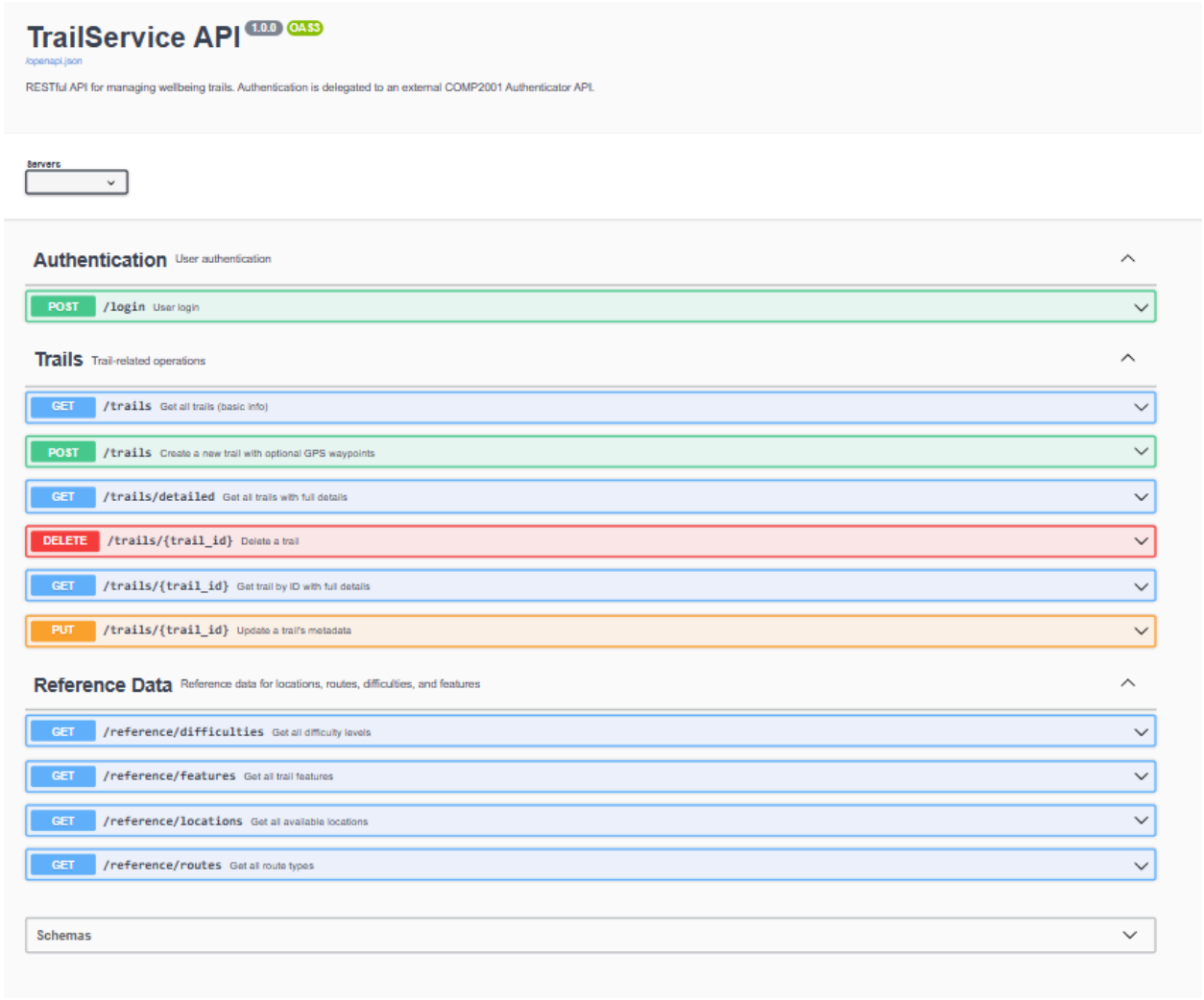


Figure 7: Swagger UI

In addition to Trail CRUD operations, the API provides reference data endpoints implemented in reference.py. These read-only endpoints allow clients to query valid values for trail creation (Appendix A).

Marshmallow schemas (Figure 8) validate incoming JSON payloads and control serialized responses, ensuring that clients cannot modify restricted fields such as timestamps or ownership data.

```python
class TrailSchema(Schema):
    trail_id = fields.Str(dump_only=True)  # Generated by API, not provided by user
    trail_name = fields.Str(required=True)
    description = fields.Str()
    visibility = fields.Str(validate=validate.OneOf(["full","limited"]), required=True)
    created_at = fields.DateTime(dump_only=True)
    updated_at = fields.DateTime(dump_only=True)  # Auto-generated timestamp
    location_id = fields.Str()
    length = fields.Decimal(as_string=True)
    estimated_time = fields.Decimal(as_string=True)
    elevation_gain = fields.Int()
    route_id = fields.Str(required=True)
    difficulty_id = fields.Str(required=True)
    created_by = fields.Str(required=True)

    points = fields.List(fields.Nested(TrailPointSchema), dump_only=True)
```

Figure 8: Marshmallow Schema Validation

5.4 Authentication

The TrailService delegates user authentication to an external COMP2001 Authenticator API (Figure 9). This separation of concerns allows centralized credential management while the TrailService focuses on business logic.

login.py > login

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

AUTH_URL = "https://web.socem.plymouth.ac.uk/COMP2001/auth/api/users"

def login():
    # Get JSON from incoming request
    data = request.get_json()
    email = data.get("email")
    password = data.get("password")

    if not email or not password:
        return {"error": "Missing email or password"}, 400

    # Send POST request to external auth service
    response = requests.post(AUTH_URL, json={"email": email, "password": password})

    if response.status_code == 200:
        try:
            return response.json(), 200
        except ValueError:
            return {"error": "Invalid JSON from auth server"}, 500
    elif response.status_code == 401:
        return {"error": "Invalid credentials"}, 401
    else:
        return {"error": "Authentication service error"}, 500
```

Figure 9: Authentication Code Delegating to External Service

5.5 RESTful Behaviour and Error Handling

The API follows RESTful principles with resource-based URLs and standard HTTP semantics. Appropriate status codes are returned for all operations:

- 200 OK for successful retrieval
- 201 Created for successful resource creation
- 204 No Content for successful deletion
- 400 Bad Request for invalid input
- 401 Unauthorized for authentication failures
- 404 Not Found for missing resources
- 409 Conflict for duplicate trail names
- 500 Internal Server Error for unexpected errors

Consistent JSON error responses are returned to clients, providing clear feedback without exposing internal implementation details.

6.0 Evaluation

6.1 Testing

The TrailService microservice was comprehensively tested using Swagger UI to verify correct behaviour across all CRUD operations. Testing covered both successful operations and error conditions to ensure robust error handling.

6.1.1 Trail CRUD Operations

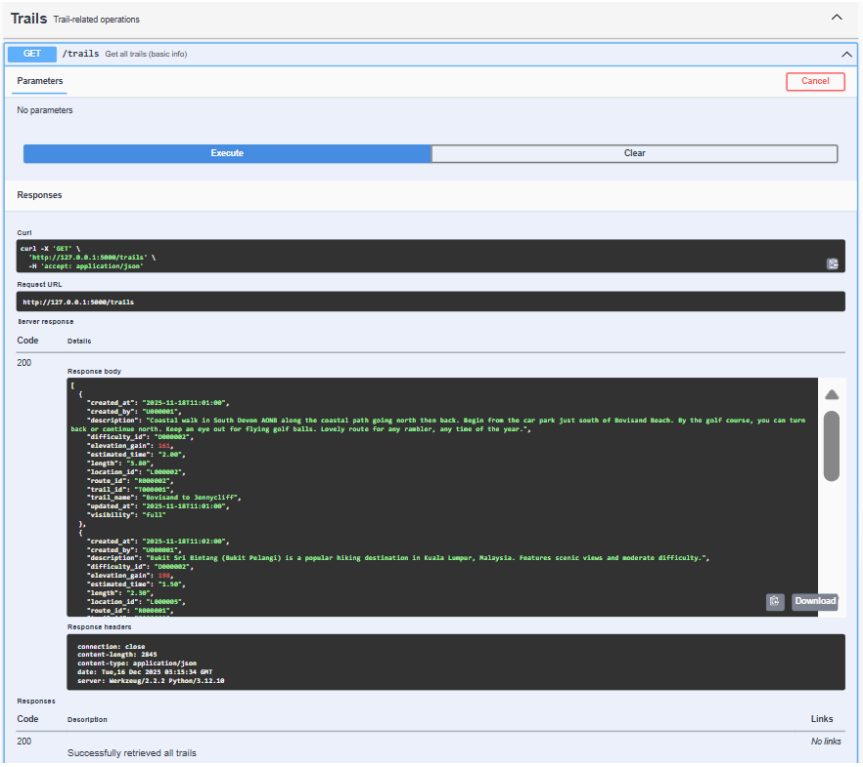


Figure 10: GET /trails

This request retrieves a list of all available trails with basic summary information.

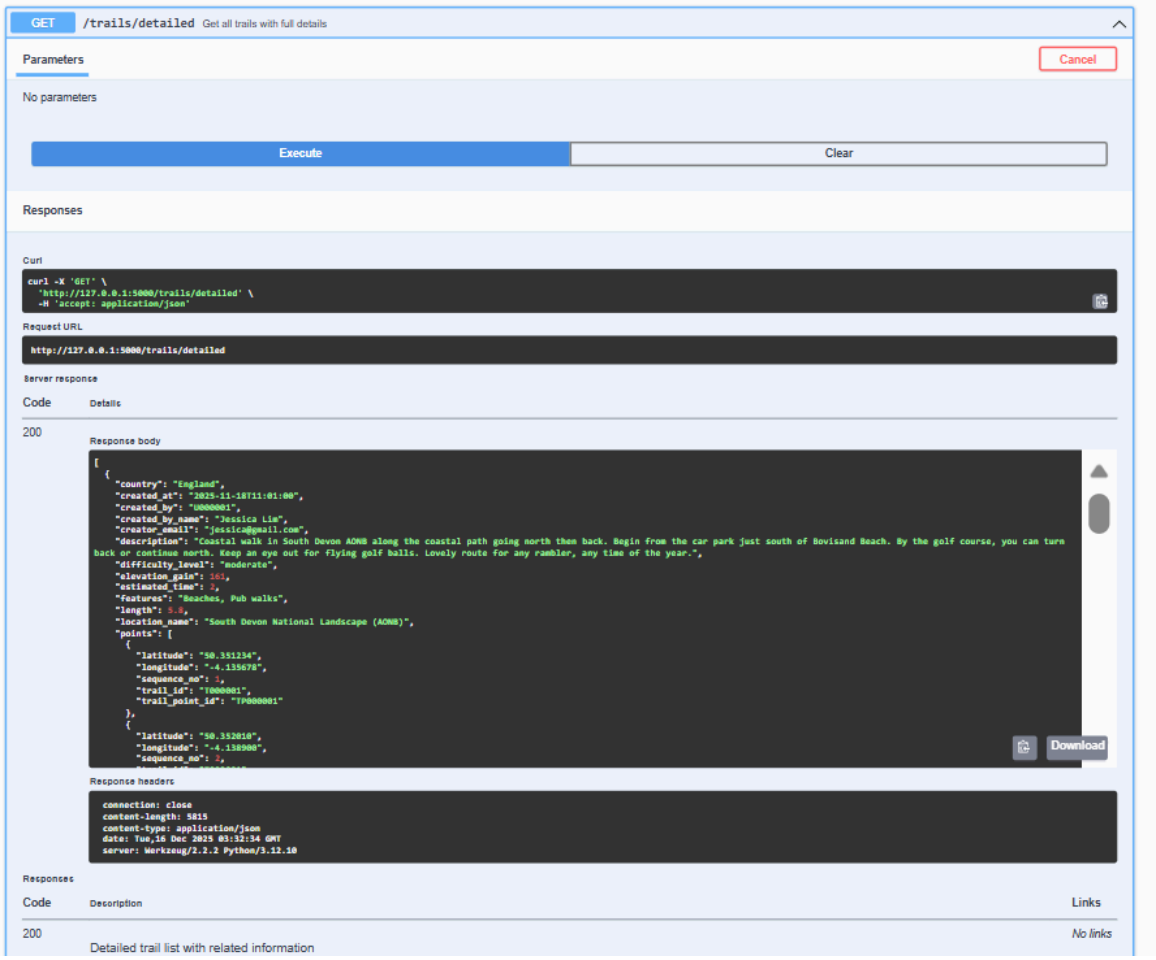


Figure 11: GET /trails/detailed

This test shows the **GET /trails/detailed** endpoint returning all trails with expanded details.

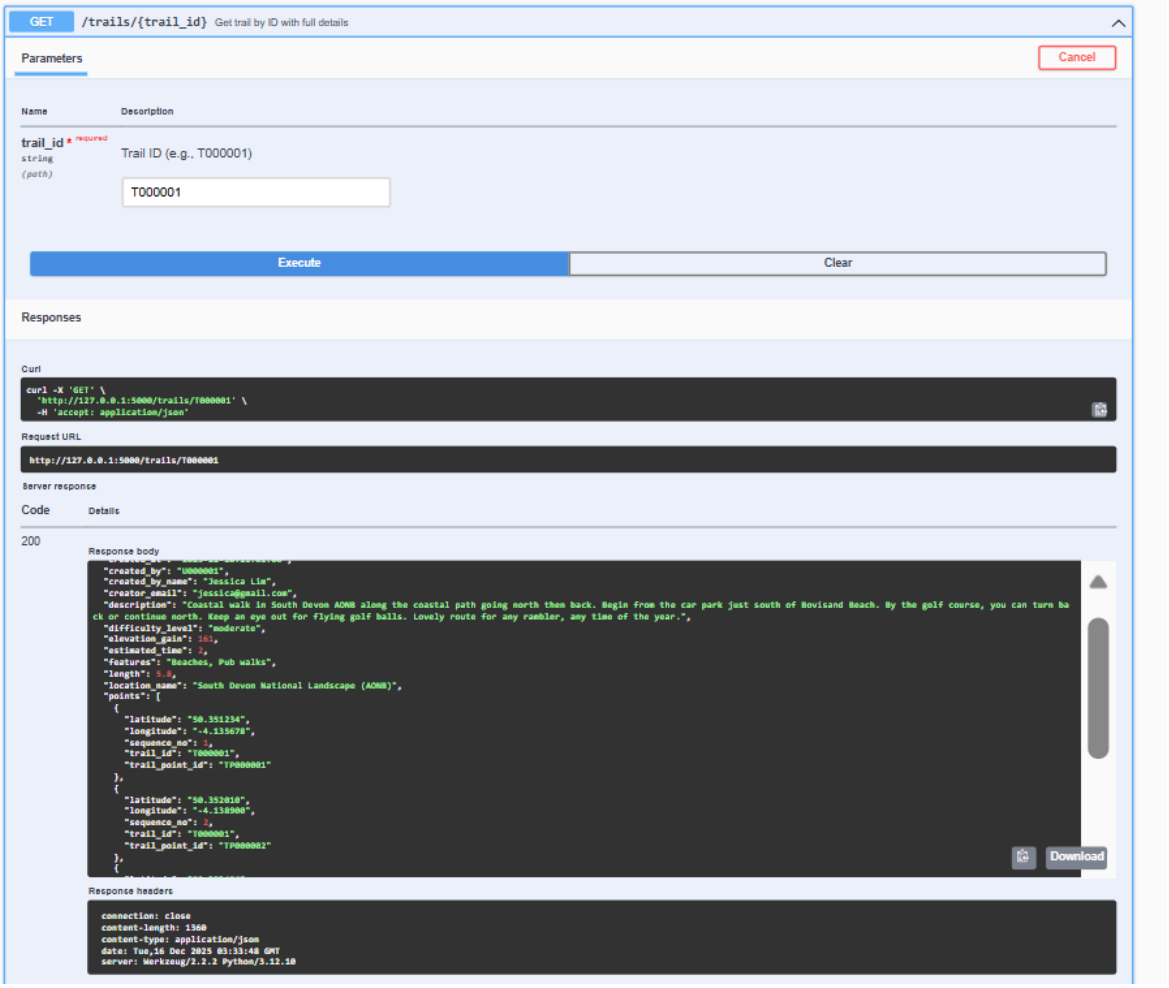


Figure 12: GET /trails/{trail_id}

This request retrieves a single trail using its unique identifier.

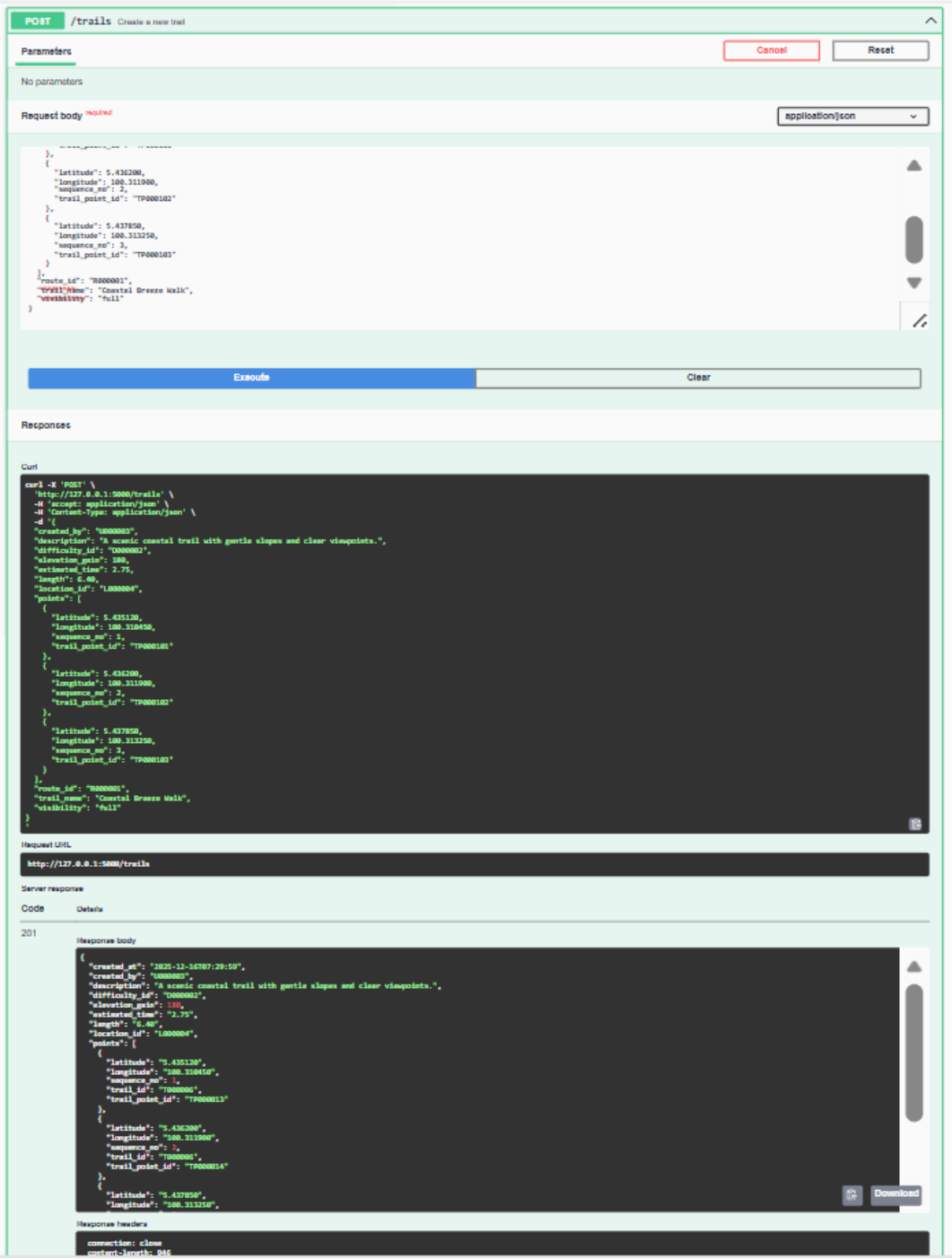


Figure 13: POST /trails – Create new trail

This request successfully creates a new trail and stores it in the database.

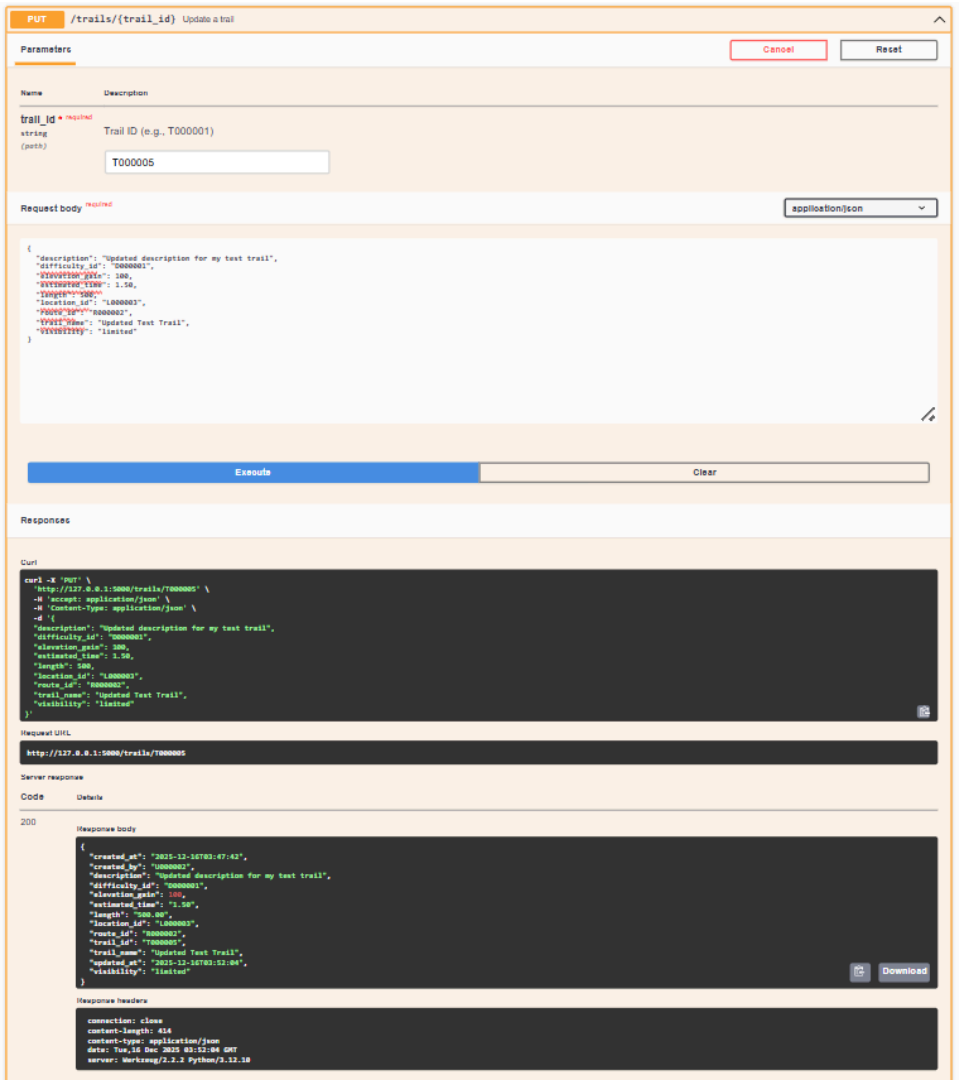


Figure 14: PUT /trails/{trail_id} – Update existing trail

This request updates the details of an existing trail.

Results Messages

	trail_id	trail_name	description	visi…	created_at	updated_at	loc…	length	esti…	eleva…	rout…	diff…	created_by
1	T000001	Bovisand to J…	Coastal walk in South Devon AONB…	full	2025-11-18 11:01:00…	2025-11-18 11:01:00…	L000002	5.80	2.00	161	R000002	D000002	U000001
2	T000002	Sri Bintang H…	Bukit Sri Bintang (Bukit Pelangi…	full	2025-11-18 11:02:00…	2025-11-18 11:02:00…	L000005	2.30	1.50	198	R000001	D000002	U000001
3	T000003	Plymbridge Ci…	Gentle circular walk through anc…	full	2025-11-18 11:03:00…	2025-11-18 11:03:00…	L000001	5.00	2.00	147	R000001	D000001	U000001
4	T000004	Kingsley Hill…	The loop trail consists of a dir…	full	2025-11-18 11:03:00…	2025-11-25 18:03:00…	L000006	3.40	1.50	185	R000001	D000002	U000001
5	T000005	Updated Test …	Updated description for my test …	limited	2025-12-16 03:47:42…	2025-12-16 03:52:04…	L000003	500.00	1.50	100	R000002	D000001	U000002

Figure 15: Database Verification for Get All Trails

The database table confirms that the trails returned by the API match the stored records, verifying data consistency between the API and the database.

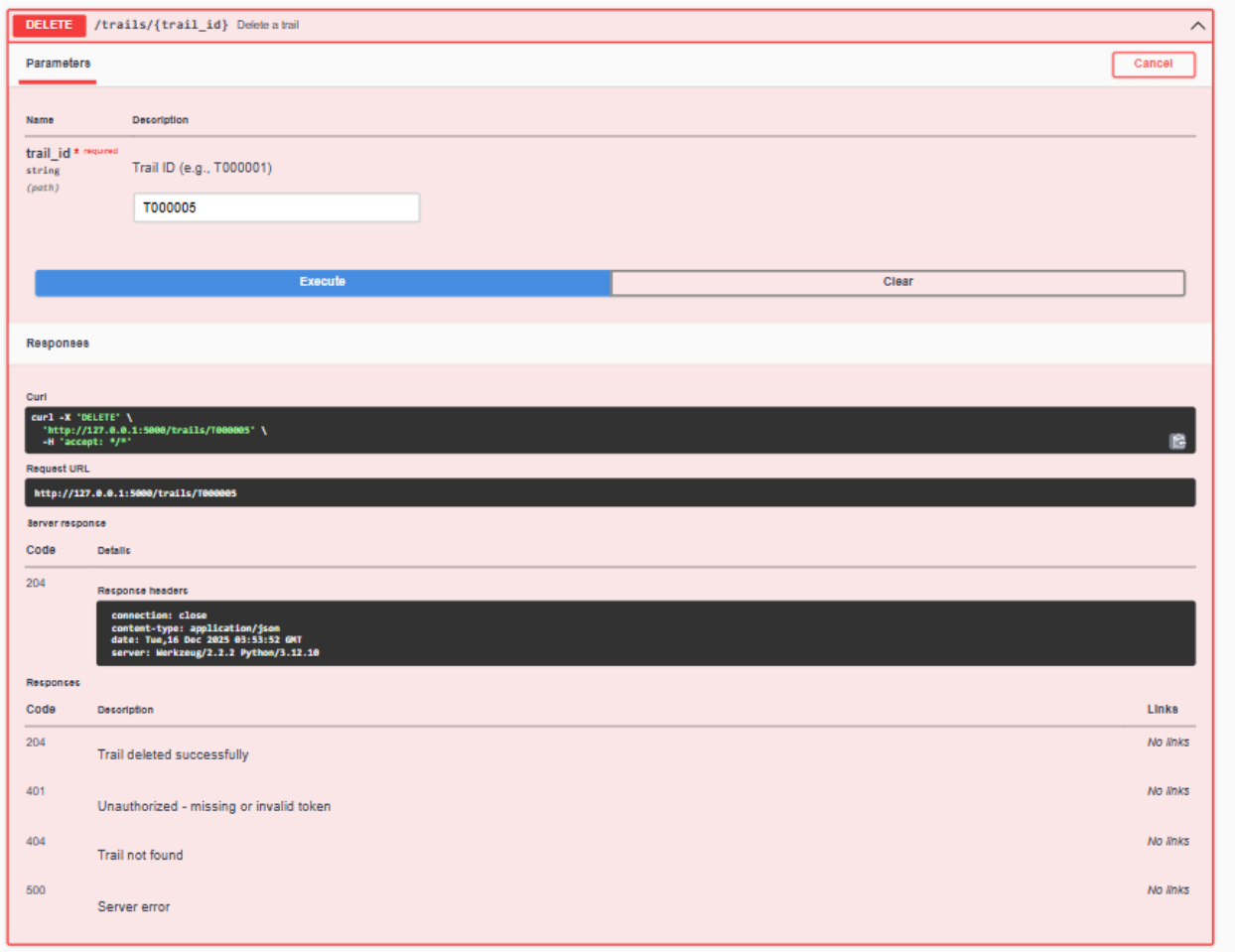


Figure 16: DELETE /trails/{trail_id} – Delete trail

This request deletes the specified trail from the system.

6.1.2 Authentication Testing

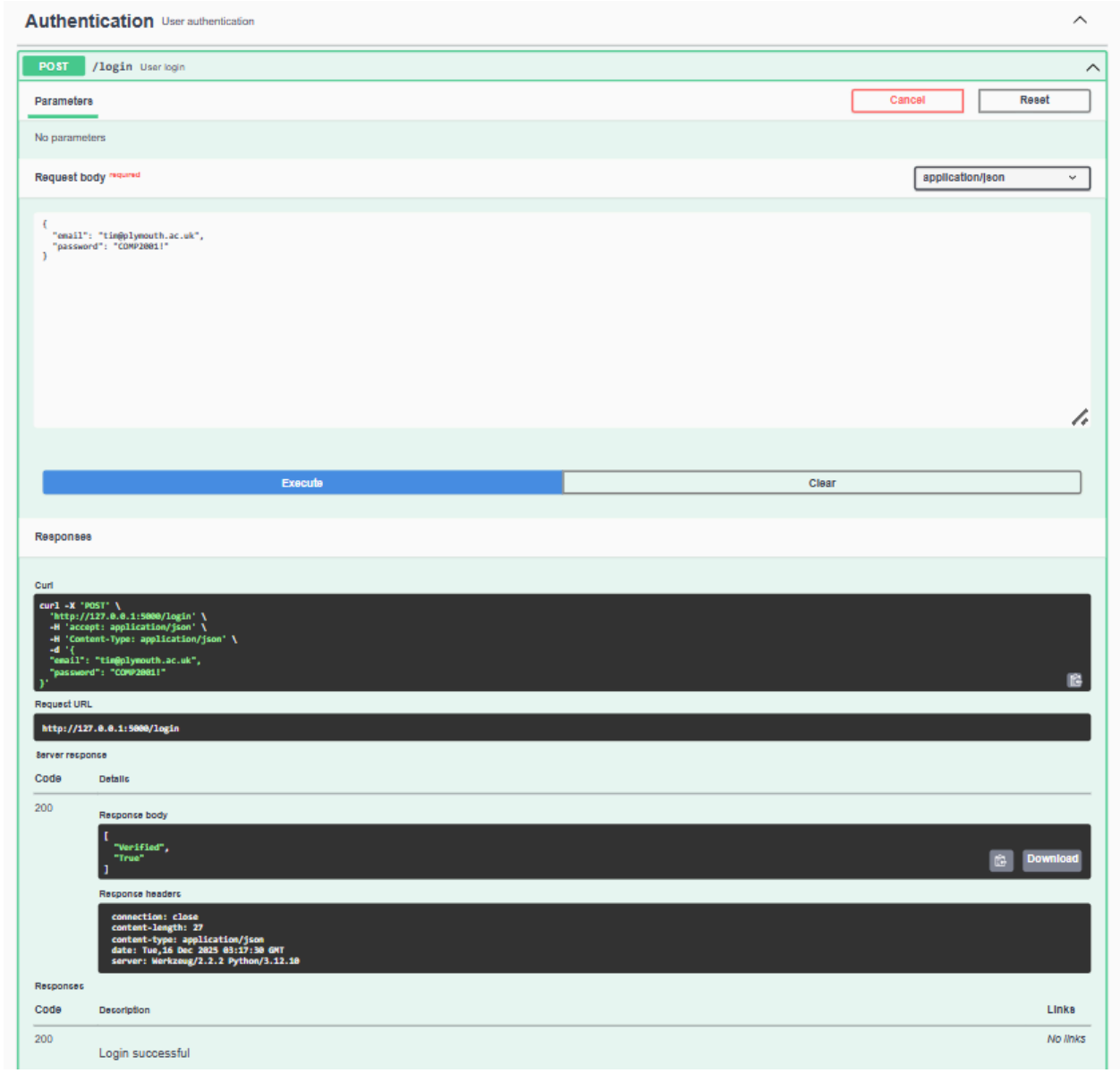


Figure 17: POST /login – Successful authentication

This request verifies valid user credentials and returns a successful login response.

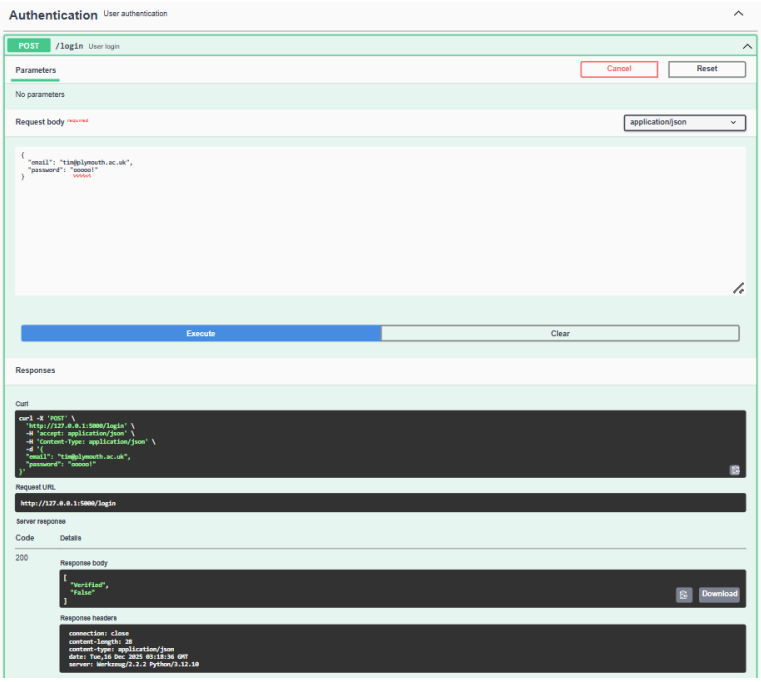


Figure 18: POST /login – Failed authentication

This request demonstrates authentication failure when incorrect credentials are provided.

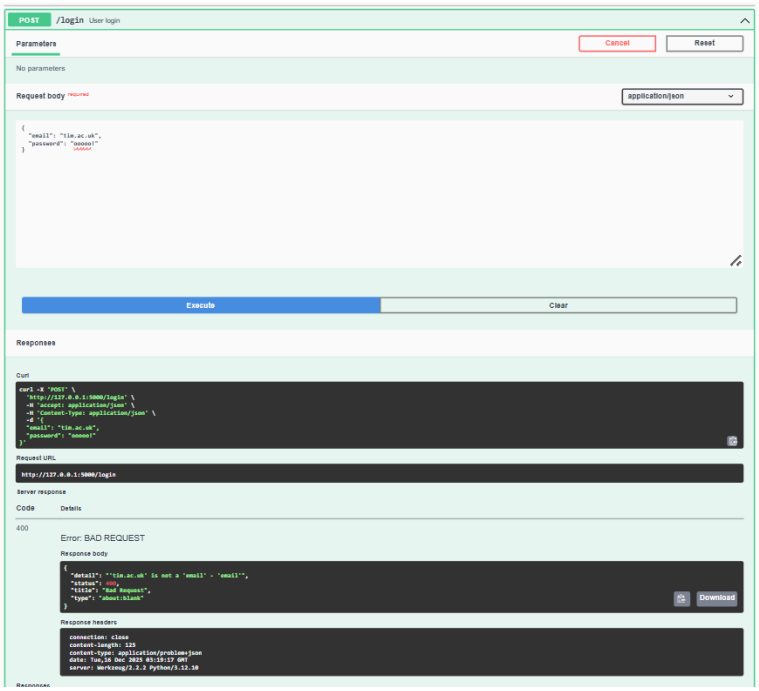


Figure 19: POST /login – Bad request handling

This request shows input validation rejecting an invalid login request.

6.1.3 Error Handling Testing:

Error handling was tested to ensure the API returns appropriate HTTP status codes and messages for invalid scenarios. Requests for non-existent resources correctly returned **404 Not Found** (Figure 20). Invalid request payloads resulted in **400 Bad Request** responses (Figure 21), while unexpected server-side failures returned **500 Internal Server Error** (Figure 22).

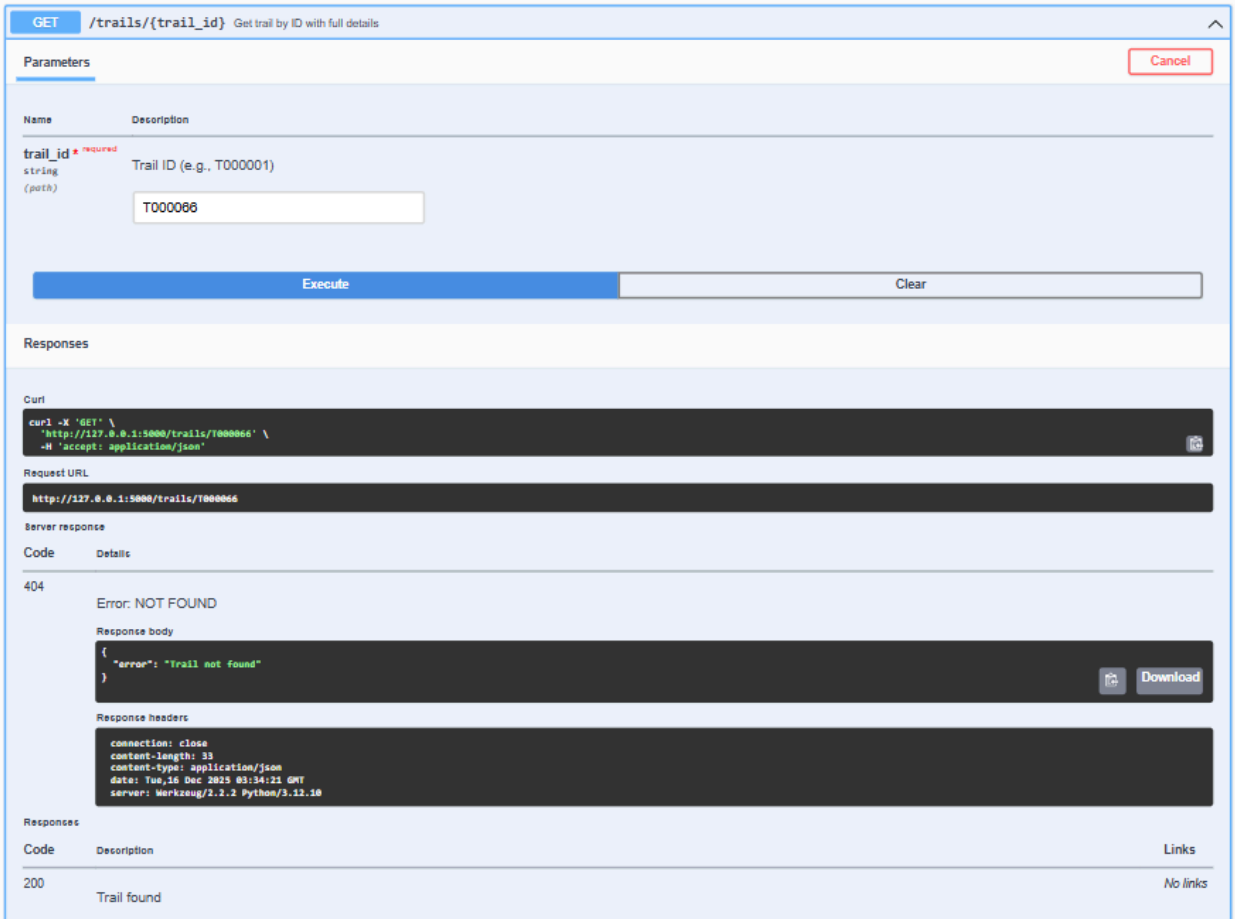


Figure 20: GET /trails/{id} – 404 response for non-existent trail

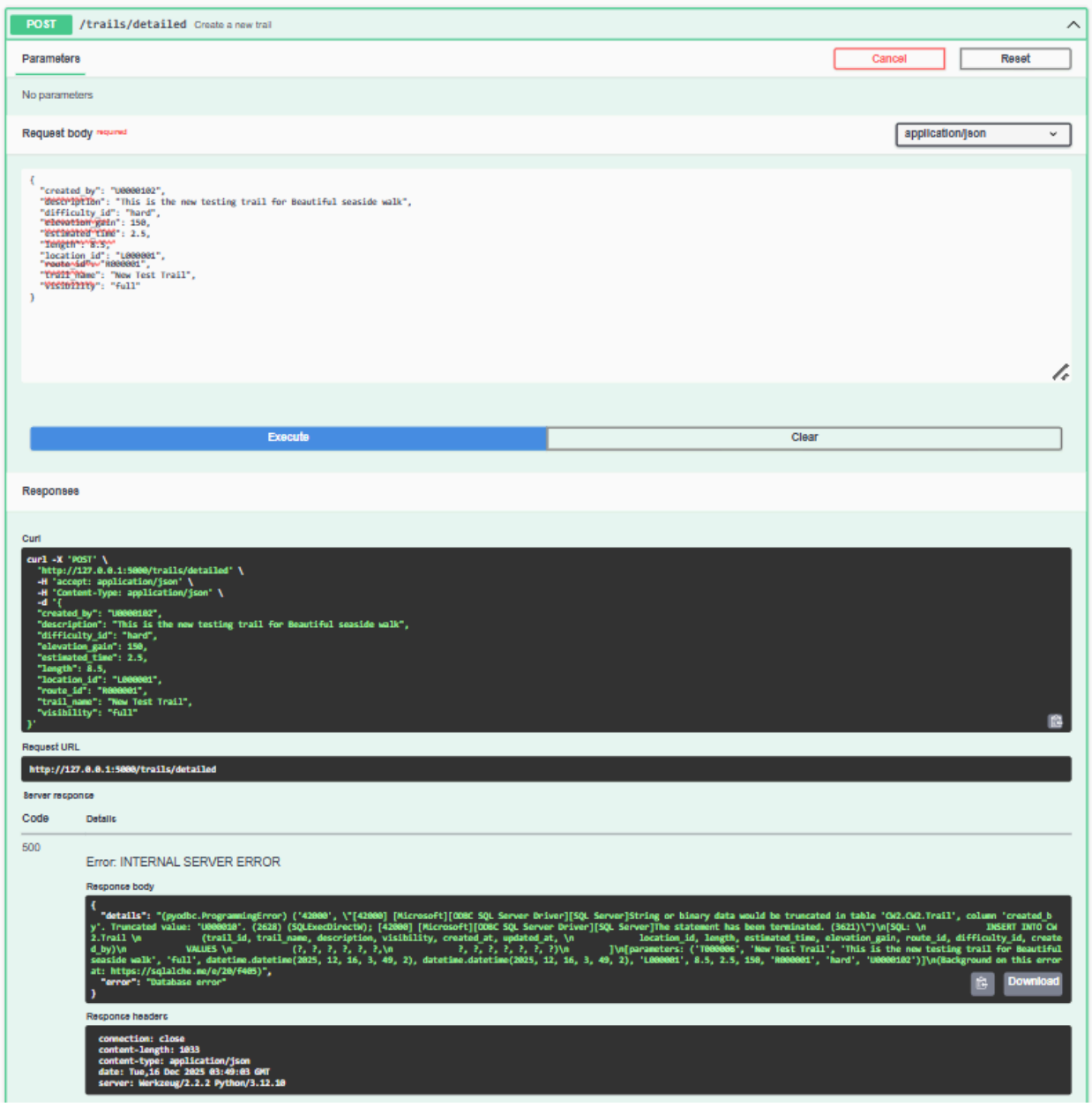


Figure 21: POST /trails – 500 Internal Server Error response

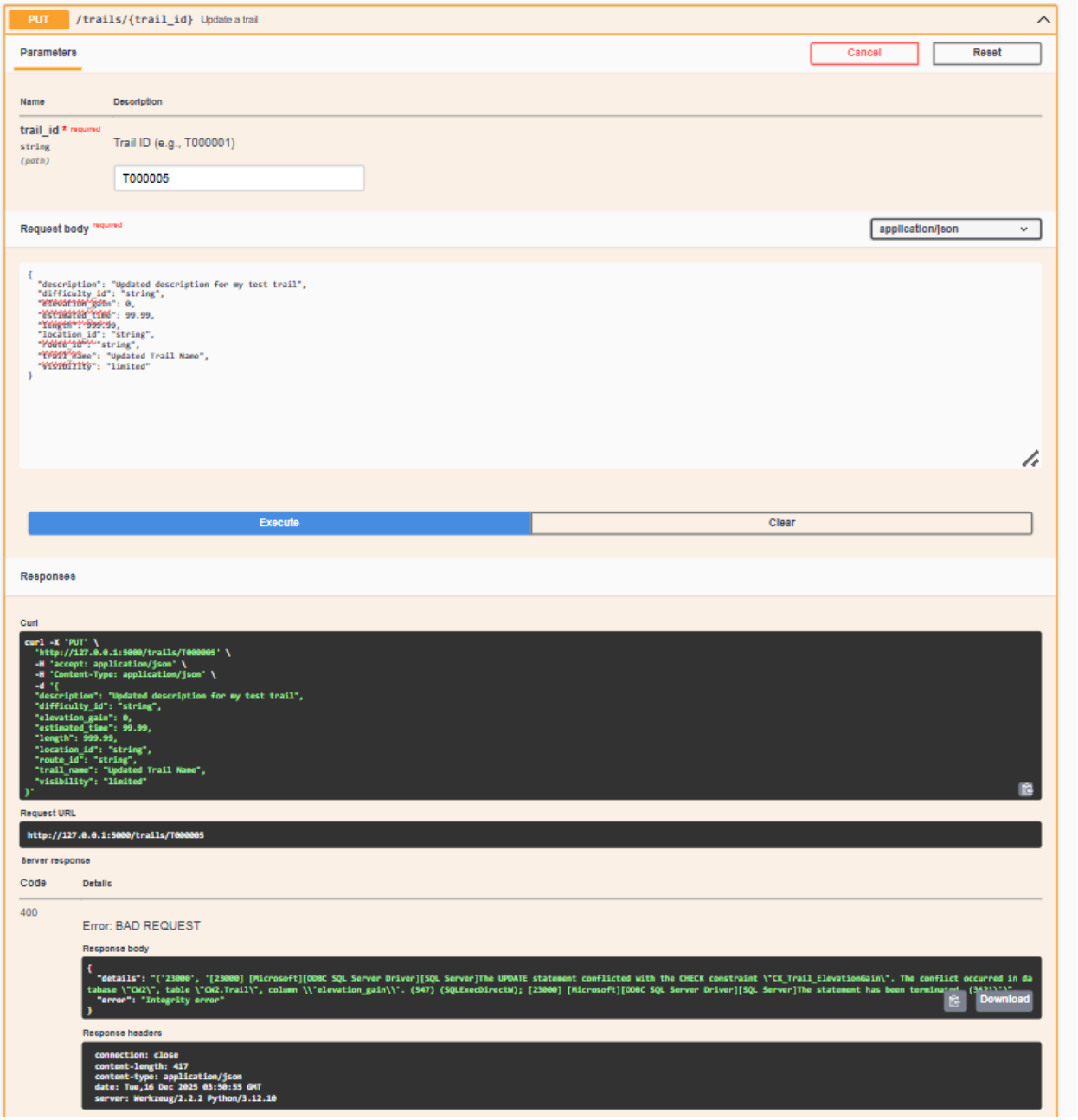


Figure 22: PUT /trails – 400 response for invalid request body

All tests confirmed that the API adheres to RESTful principles by providing correct status codes, consistent response formats, and clear error messages.

6.2 Strengths & Weaknesses

The TrailService microservice uses a clear modular architecture with well-defined RESTful endpoints documented through OpenAPI and Swagger UI. The database schema follows Third Normal Form, reducing redundancy and enforcing referential integrity. Trail ownership is enforced by associating trails with creator identifiers, ensuring accountability for create, update, and delete operations, while SQL triggers provide reliable audit logging.

Reflecting on the implementation, several limitations were identified. The current API returns complete trail datasets without pagination or filtering, which may affect performance as data volume grows. In addition, authentication relies entirely on real-time calls to the external service, which may introduce latency and reduce resilience if the service becomes unavailable.

6.3 Further Work

Future development would focus on improving scalability and robustness. Key enhancements include adding pagination and query filtering, as well as introducing token caching and expiry checks to reduce reliance on the external authentication service. If supported, token-based session management and role-based authorisation could further strengthen access control. Additional improvements may include role-based permissions, user activity history, trail ratings, and expanded automated testing to enhance reliability and long-term maintainability.

References

ACM (2018) ACM Code of Ethics and Professional Conduct. Available at: https://www.acm.org/code-of-ethics (Accessed: 10 December 2024).

BCS (2022) BCS Code of Conduct. Available at: https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/ (Accessed: 10 December 2024).

Digital Preservation Coalition (2015) Digital Preservation Handbook. 2nd edn. Available at: https://www.dpconline.org/handbook (Accessed: 2 December 2024).

European Parliament (2016) Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation). Official Journal of the European Union, L119, pp. 1-88. (Accessed: 11 December 2024).

Microsoft (2023) SQL Server Transaction Log Architecture and Management Guide. Available at: https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-log-architecture-and-management-guide (Accessed: 28 November 2024).

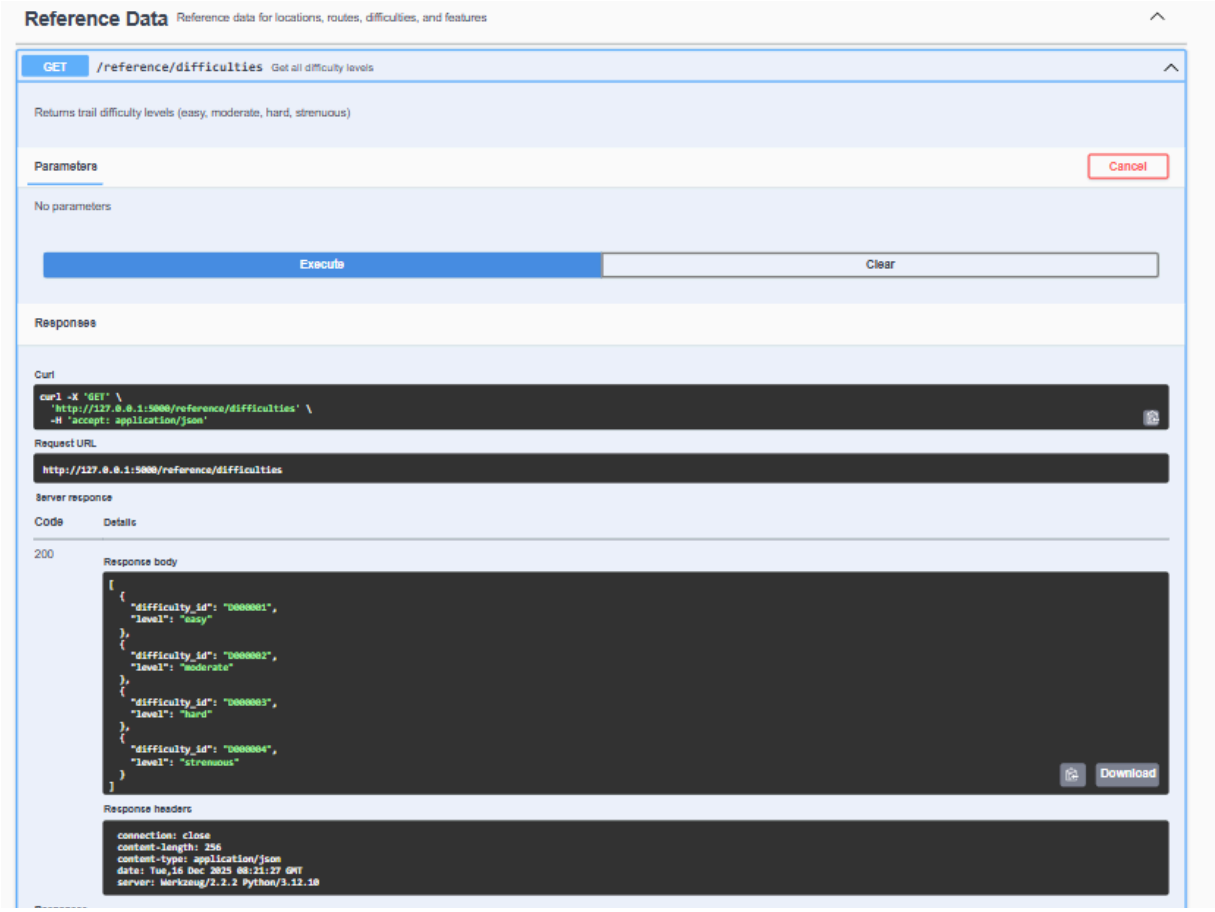
NIST (2013) Security and Privacy Controls for Federal Information Systems and Organizations. NIST Special Publication 800-53, Revision 4. Gaithersburg, MD: National Institute of Standards and Technology. doi: 10.6028/NIST.SP.800-53r4.

OWASP (2021) OWASP Top Ten 2021. Available at: https://owasp.org/www-project-top-ten/ (Accessed: 11 December 2024).

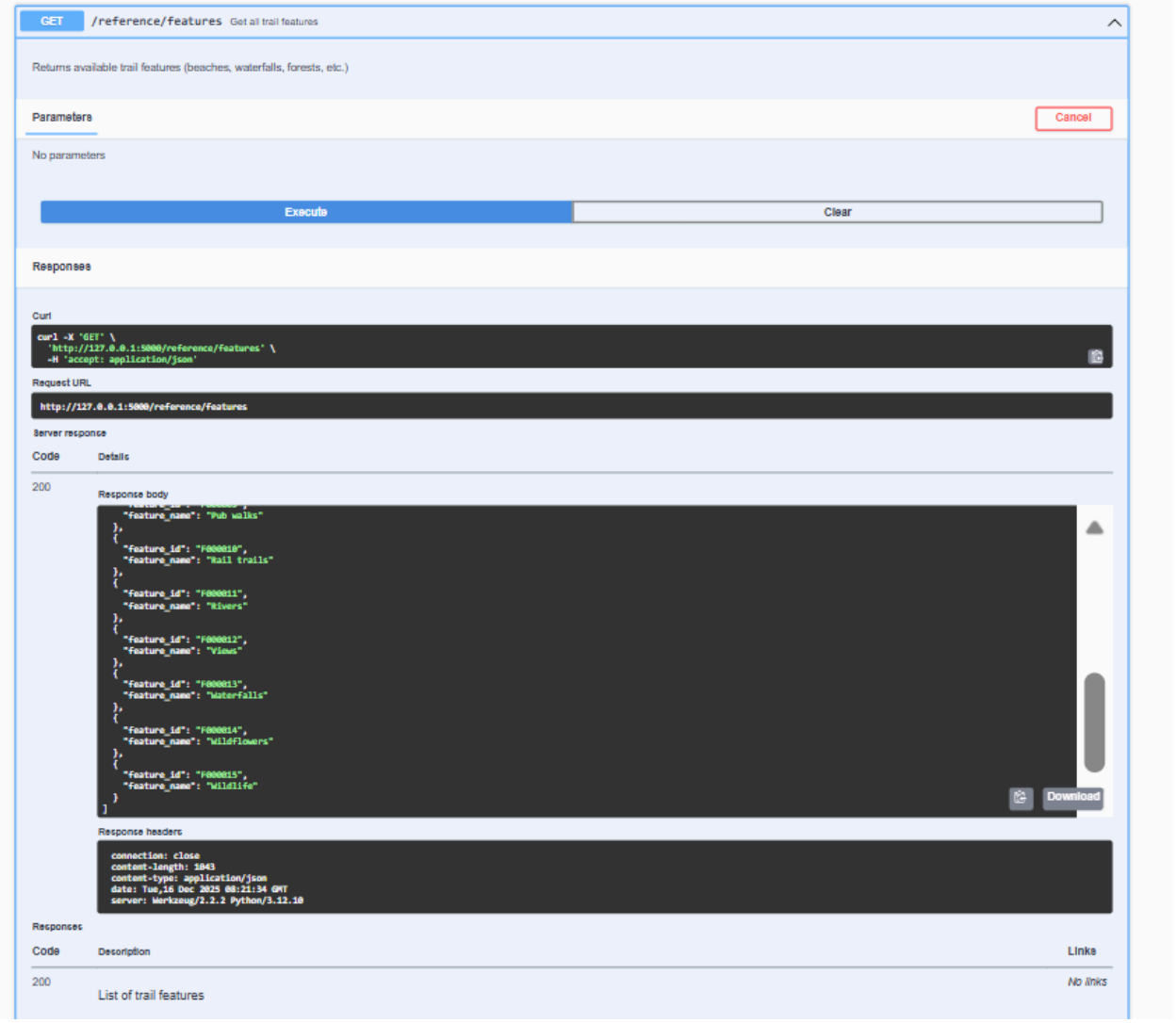
Pfleeger, C.P. and Pfleeger, S.L. (2015) Security in Computing. 5th edn. Upper Saddle River, NJ: Prentice Hall.

Sling Academy (2024) Preventing SQL Injection in SQLAlchemy: Best Practices. Available at: https://www.slingacademy.com/article/preventing-sql-injection-in-sqlalchemy/ (Accessed: 10 December 2024).

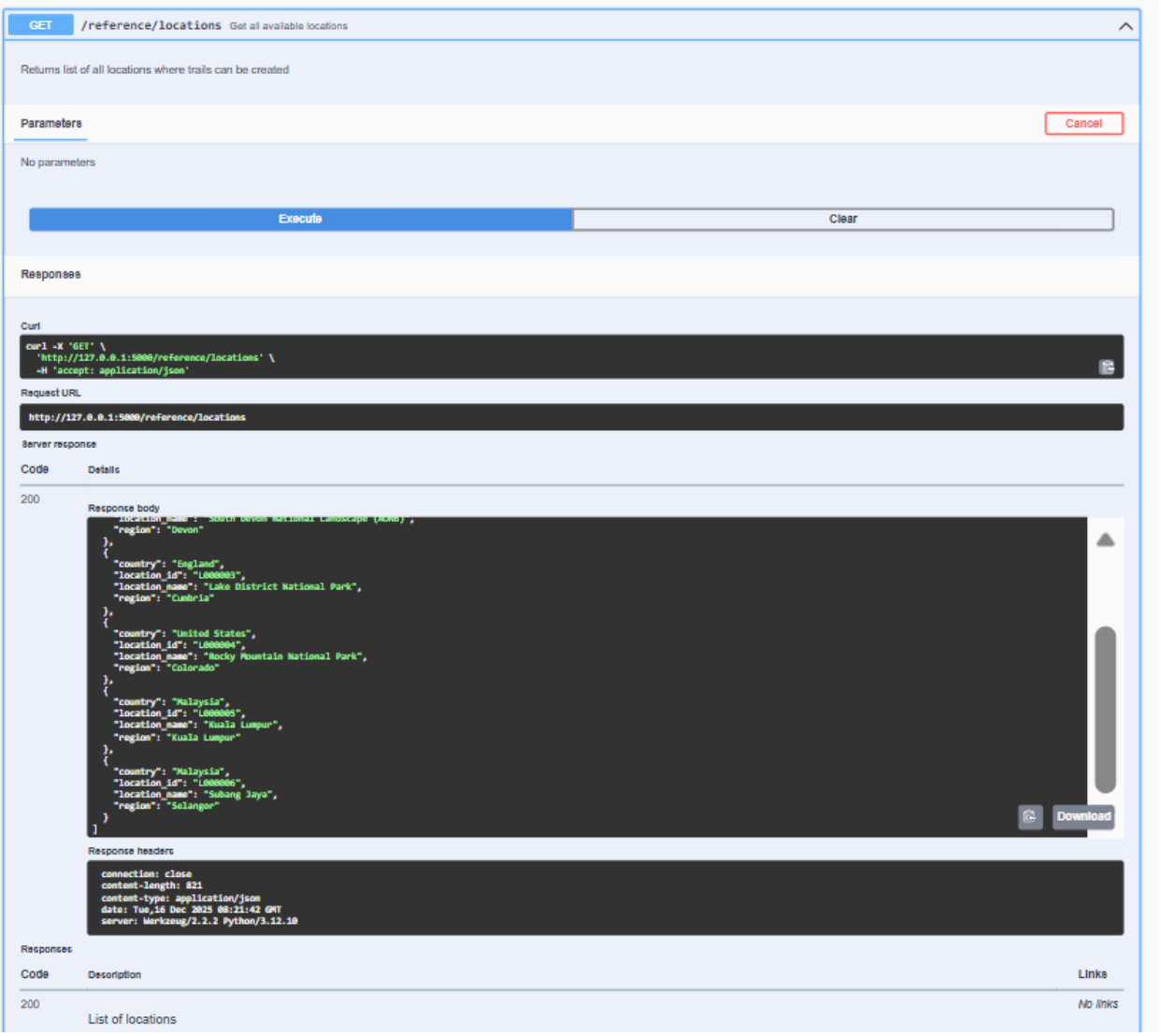
Appendix A: Reference Data Endpoints



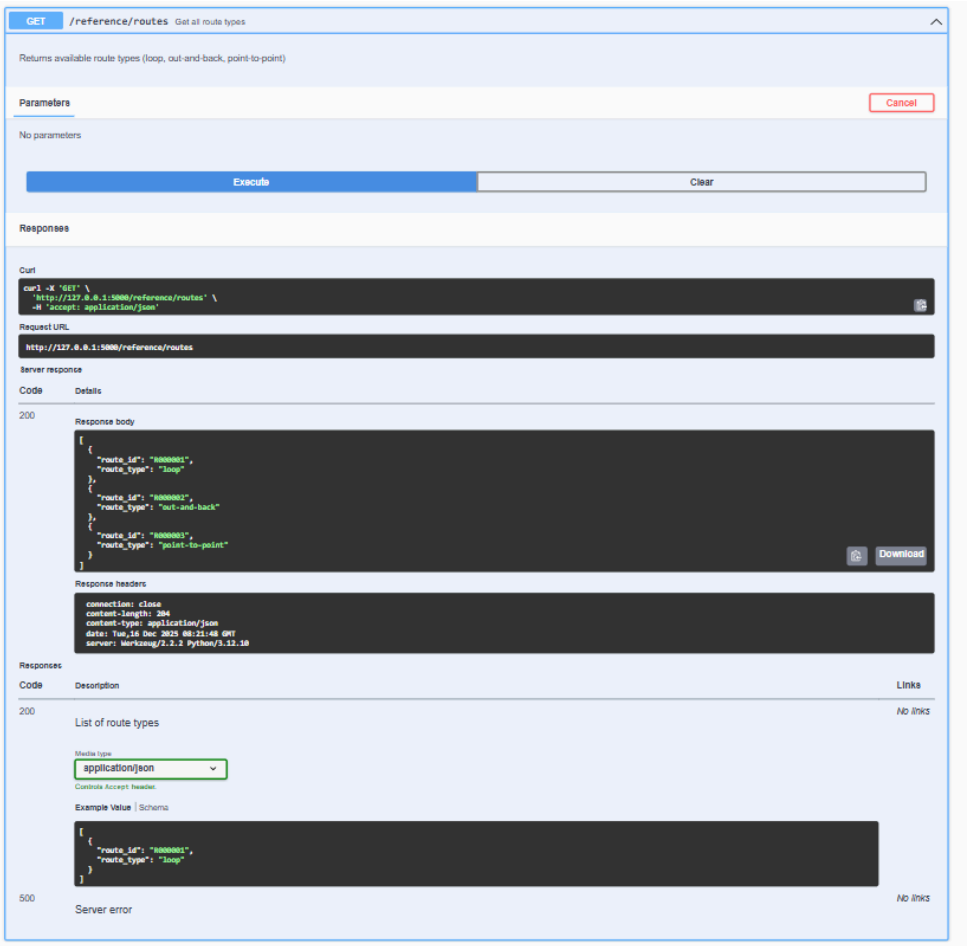
Appendix 1: GET /reference/difficulties - Successfully Retrieved All Difficulty Levels



Appendix 2: GET /reference/features - Successfully Retrieved All Trail Features



Appendix 3: GET /reference/locations - Successfully Retrieved All Locations



Appendix 4: GET /reference/routes - Successfully Retrieved All Route Types

Action	HTTP Verb	URL Path	Description
Read difficulties	GET	/reference/difficulties	Retrieve a list of all trail difficulty levels (read-only reference data)
Read features	GET	/reference/features	Retrieve all supported trail features for descriptive use
Read locations	GET	/reference/locations	Retrieve available trail locations
Read routes	GET	/reference/routes	Retrieve available route types

Table 3: Reference Data Endpoints

This module provides read-only endpoints for querying reference data used in trail creation, including available locations, route types, difficulty levels, and trail features (Table 3).