COMP2001 – CW2

Introduction

This document provides an overview of the Trail microservice developed. In this report, key design elements, legal and ethical considerations, implementation details, and evaluation of the microservice are presented. The trail microservice is used to allow users to explore various trails, manage their details and track the features associated with each trail. It utilizes a RESTful API for users and admins to interact with the system. The design and implementation features supporting documents including, UML diagrams and code samples to illustrate the design to deployment process.

The microservice manages the trail data, ensuring secure and reliable access through the RESTful API. It is capable of performing CRUD (Create, Read, Update and Delete) operations on trail-related data. The API interacts with an SQL database to store and retrieve data efficiently. It also integrates an external authenticator API to handle user authentication, by verifying user credentials and issuing tokens to give users access. The microservice was deployed using docker and managed via a version-controlled Github repository. This repository contains all the source code for the project, ensuring that it can be easily updated and maintained. Links to these resources are provided below:

Github: The work is under COMP2001/CW2/comp2001_flask
https://github.com/Mattrfish/COMP2001.git

Dockerfile: mattrfish/2001_flask

Background

The trail application is designed to encourage users to explore outside by providing them with access to different trails for hiking, biking, or any other outdoor activities. The goal of app is to help users discover new trails that maybe looking for, e.g. a user may want to go biking and can look for the bike trails. In support of this goal, the trail microservices role is used to store, manage and provide access to the trail-related data that the users will interact with.

The microservice is responsible for storing and processing all the information related to trail, so that the users can view trail data efficiently. It provides the following core functionality:

- CRUD operations: The service allows for creating, reading, updating and deleting trail data.
- User Authentication: By integrating the authentication API, the service ensures that only validated users can access trail data. Users can view trial data, but only admins can modify the data.
- Trail ownership: Trails are owner by users, and a trail can have multiple features.

The application architecture follows the RESTful API design, ensuring that the data being exchanged is in the JSON format. The microservice interacts with the SQL database, where the trail information is stored securely.

Key Components:

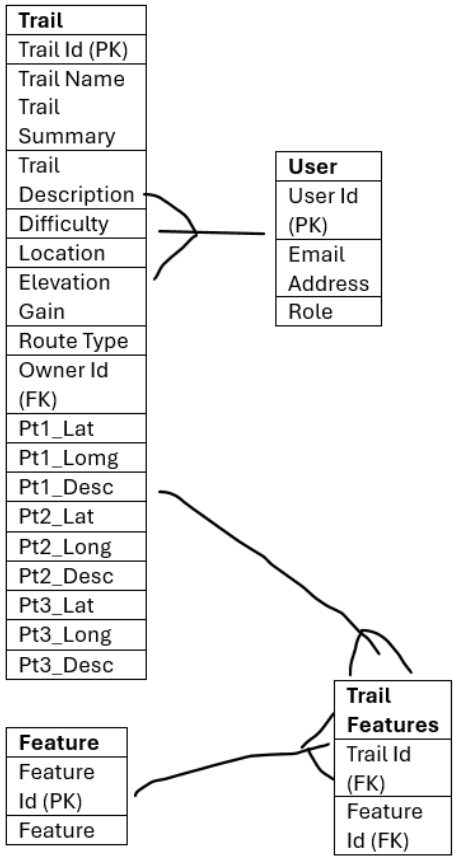
1. Trails: This includes details such as difficulty, location, length and elevation gain.
2. Features: These are specific details of the trail like ponds, dirt paths, benches etc.
3. Users: Both general users and admins interact with the service based on their role, with admins having full CRUD access to the trail data.

Design

The design of the trail microservice follows the principles of RESTful architecture. The ERD and UML diagrams below illustrate the systems structure:

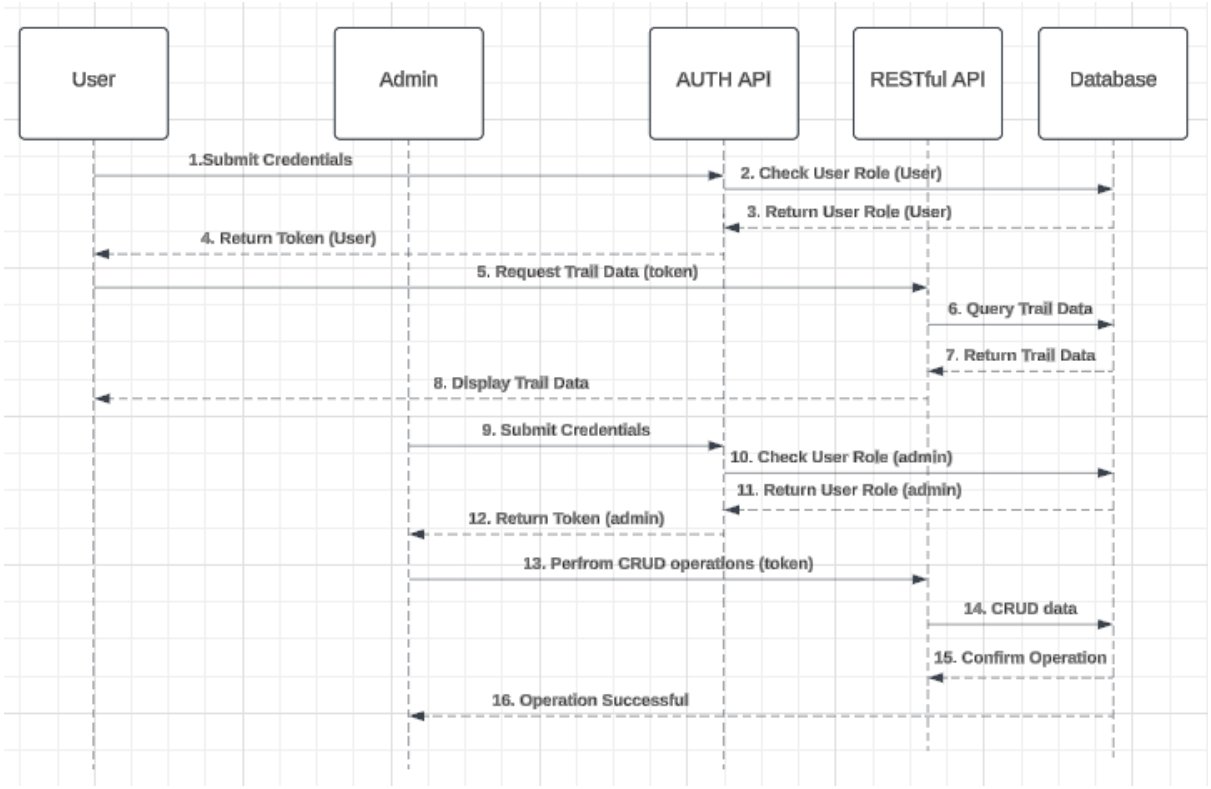
ERD

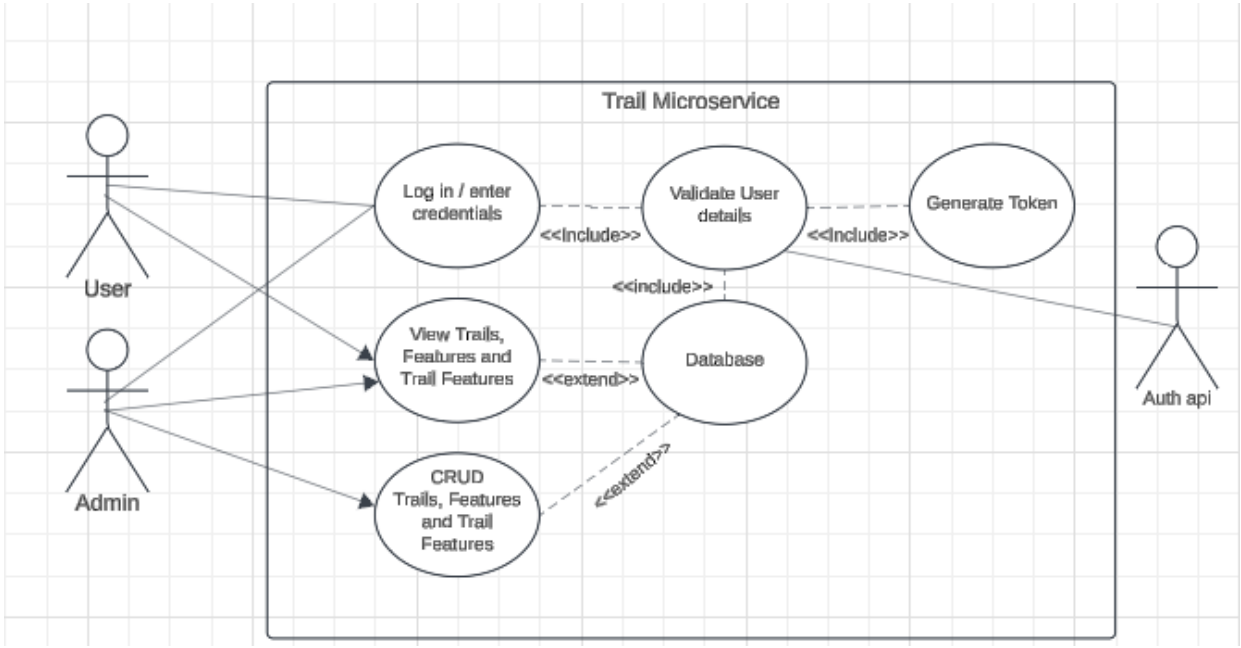
The ERD below shows the tables, their entities and the relationships between them. The Trail and features table uses a junction table to remove the many-many relationship. Each trail can have many features, and those features can also be in many different trails. Each trail can be owned by one user, but one user can own many different trails.



UML

The UMLs below show the user cases; how the users will interact with the microservice. The users will enter their credentials which will be sent to the authenticator API. It will validate those credentials from the API and then validate them from the database to generate a specific token depending on the users role (admin or user). The users can view trails, features and trail features and the admins can CRUD trails, features and trail features.





User Types

User: A general user who can read trails, features and trail features.

Admin: An admin with full CRUD access for trails, features and trail features.

Use Cases

Authenticate:

- User Type: User, Admin
- Description: The user or admin sends credentials to the authentication API, which returns an access code and the RESTful API link.
- Precondition: Valid credentials must be provided.
- Postcondition: Successful login provides access to appropriate API functionality based on the user role.

Read Trails, Features or Trail Features:

- User Type: User, Admin
- Description: Users and admins can view all trail information
- Precondition: The user must be authenticated and enter their access code.
- Postcondition: Trail data is displayed from the database

CRUD Trails, Features or Trail Features:

- User Type: Admin
- Description: Admins can create, read, update, and delete trail information.
- Precondition: The admin must be authenticated and enter their access code.
- Postcondition: Trail information is modified in the database.

Legal, Social, Ethical and Professional

When developing the RESTful API, I had to ensure that it adhered to legal, social, ethical and professional standards, with a focus on addressing risks outlined by the OWASP top 10.

Legal Considerations

- Data Protection: The microservice complies with the GDPR. The user credentials are stored securely within the authentication API. For the protection of user passwords when authenticating credentials, they are never exposed in the URLs. The data is securely sent via the POST method, avoiding the exposure of sensitive information in the request header when using the GET method.
- Authentication and session Management: Additionally, token based authentication is used to ensure that only authenticated users are gaining access to the RESTful API.
- Injection Attack: I have used prepared statements and queries to mitigate the risk of SQL injection.

Social and Ethical Considerations

- Access Control: Ensuring that users of different roles (admin and user) have access to specific permissions. This minimizes the risk of unauthorized access and manipulation of data, ensuring only users with the proper authorization can perform certain actions.
- Storage: Personal and sensitive data is stored with the consent of the user. The system doesn't store any unnecessary or excessive user data, only their roles and email addresses.

Professional Standards

- Quality: The code must meet industry standards, following best practices for readability, structure and maintenance. It is version-controlled using GitHub, with clear documentation and a README file.
- Testing: It has been tested for functionality and security to ensure that it works as expected and that OWASP security concerns are mitigated effectively.

Implementation

I created the microservice using YML, HTML and python and I have deployed it on docker.

Trail Management

The microservice provides endpoints to create, read, update, and delete trail data.

The endpoints used are:

- /trails
- /trails/{Trailid}
- /features
- /features/{FeaturesId}
- /trails/{TrailId}/features
- /trails/{TrailId}/features/{FeaturesId]

I will provide an example of the read_all() function for a trail:

```
# Function to read all trails (Open to all)
def read_all():
    # Extract token from query parameter
    token = request.args.get("auth")
    if not token:
        return jsonify({"message": "Authorization token required"}), 401

    try:
        # Decode the token manually
        decoded_token = decode_token(token)
        identity = decoded_token.get("sub")  # Extract 'sub' field (user detail
        if not identity:
            return jsonify({"message": "Invalid token"}), 401

       # Parse the 'sub' field as a dictionary
        user_data = json.loads(identity)
        user_id = user_data.get("id")  # Extract the UserID
        role = user_data.get("role")  # Extract the role

        # Verify user exists in the database
        user = User.query.filter_by(UserID=user_id).first()
        if not user:
            return jsonify({"message": "User not found"}), 404

    except Exception as e:
        return jsonify({"message": f"Token decoding error: {str(e)}"}), 401

    # Fetch and return trails
    trails = Trail.query.all()
    return trails_schema.dump(trails), 200
```

The user will enter their code into the parameter on the swagger ui (like they will enter the trailid) . The code is received and decoded using the deocde_token() function, to extract the user information (id, email and role). If the token is invalid or the user does not exist in the database, it returns an error message. Once the token is validated, the function queries the database for all the trail and returns them using a JSON format. The authentication is done for all CRUD functions.

User Authentication

The application integrates the authentication api for handling the user authentication:

```
# authenticate userwith api, then retrieve the user role from db
def authenticate_user(email, password):
    credentials = {"email": email, "password": password}
    try:
        response = requests.post(AUTH_API_URL, json=credentials) #POST request to auth api
        if response.status_code == 200:
            response_data = response.json()
            if response_data and response_data[1] == "True": # verify successful auth
                user = User.query.filter_by(EmailAddress=email).first() #get user details from db using email
                if user:
                    return {"email": email, "role": user.Role, "id": user.UserID} # return user details
        return None
    except Exception as e:
        print(f"Error with authentication API: {e}")# error message
        return None

#hanlde login and return an access token for auth users
@app.route("/swagger", methods=["POST"])
def swagger_ui():
    data = request.json
    email = data.get("email")
    password = data.get("password")# get details from the request

    if not email or not password:
        return jsonify({"message": "Email and password are required"}), 401

    user = authenticate_user(email, password) # auth the user and retrieve the details
    if user:
        #create an access token for auth user
        access_token = create_access_token(identity=json.dumps(user))
        print(F"Access Token: {access_token}")
        redirect_url = f"http://localhost:8000/api/ui"
        print(F"Redirecting to: {redirect_url}")

        #return access token and ui url
        return jsonify({"access_token": access_token, "redirect_url": redirect_url}), 200
    else:
        return jsonify({"message": "Invalid credentials"}), 403
```

The authenticate_user() function authenticates users by sending a POST request with the provided email and password to the auth api (for example: curl -X POST http://localhost:8000/swagger -H "Content-Type: application/json" -d "{\"email\": \"tim@plymouth.ac.uk\", \"password\": \"COMP2001!\"}"). If its successful, it retirevies the users' details from the database (using their email). If it fails, it returns None. The /swagger route handles user login by accepting email and password from the request, calling the authenticate_user() function, and, if successful, creating an access token for the user. The access token and the swagger ui URL are then returned. If it fails it responds with "Invalid Credentials"

Database Integration

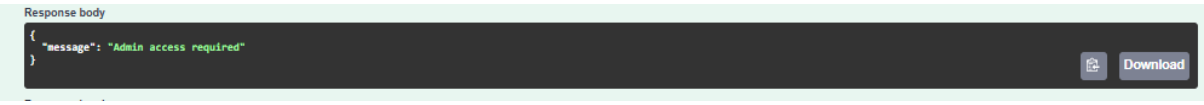
The data is stored on SQL server and is designed to manager user information, trails and features. The config.py file connects the application to the correct database server.

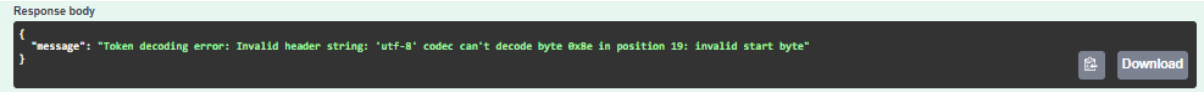
```python
#config.py

import pathlib
import connexion
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
import os

basedir = pathlib.Path(__file__).parent.resolve()
connex_app = connexion.App(__name__, specification_dir=basedir)

app = connex_app.app
app.config["SQLALCHEMY_DATABASE_URI"] = (
    "mssql+pyodbc:///?odbc_connect="
    "DRIVER={ODBC Driver 17 for SQL Server};"
    "SERVER=DIST-6-505.uopnet.plymouth.ac.uk;"
    "DATABASE=COMP2001_MFish;"
    "UID=MFish;"
    "PWD=UjkN272*;"
    "TrustServerCertificate=yes;"
    "Encrypt=yes;"
)
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

app.config["JWT_SECRET_KEY"] = "123"

db = SQLAlchemy(app)
ma = Marshmallow(app)
```

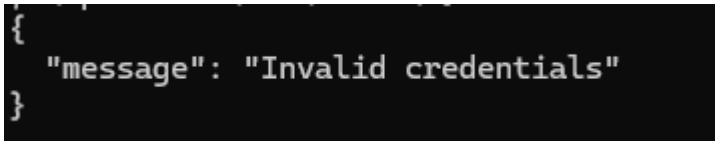
And then the models.py file gets the columns and formats of the trail table within the database, to support the operations such as creating, reading, updating and deleting trail entries.

Error Handling

The microservice checks for errors such as unauthorized access, invalid IDs, and invalid input data. Appropriate messages are returned in JSON format.

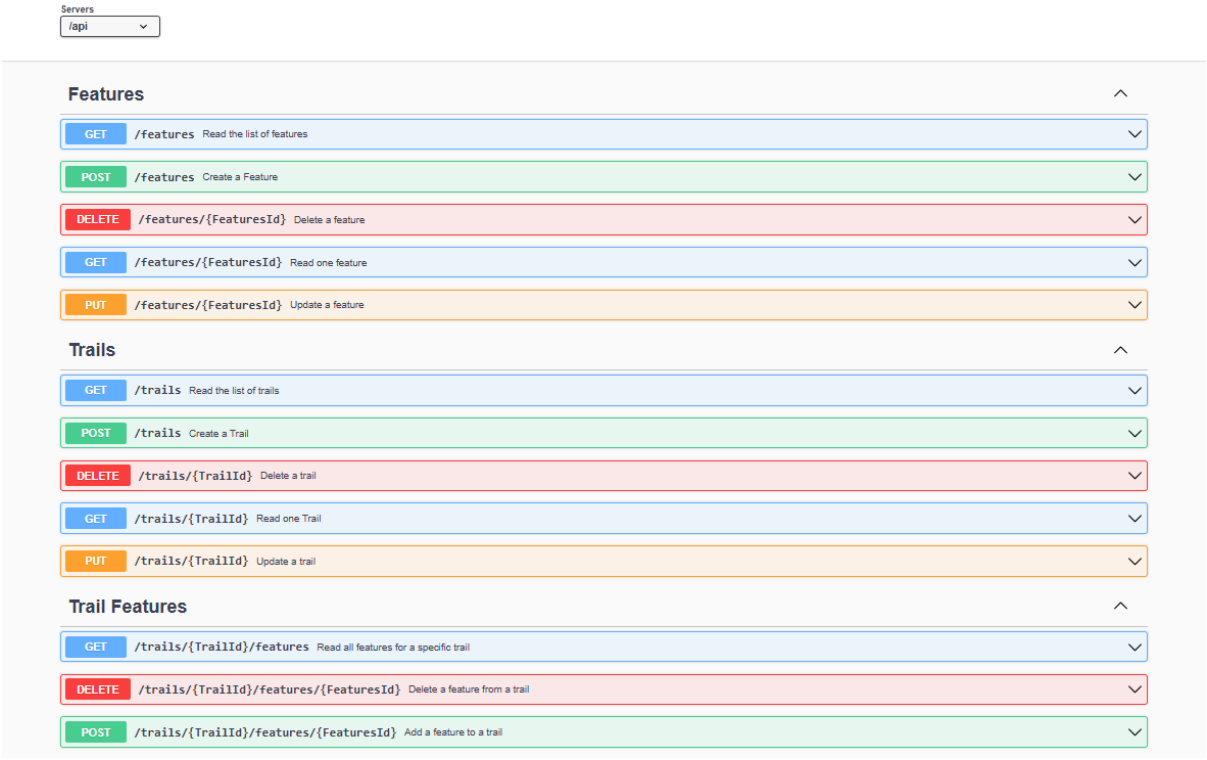






Deployment

The application is deployed on docker and managed and updated via Github, ensuring changes are tracked and versioned.



This is the swagger ui, showing all of the endpoints.

Evaluation

I have tested the microservice in order to provide that everything is successful working. I have also gathered the areas of improvement to show what needs to be improved in the future.

Testing

Authentication API

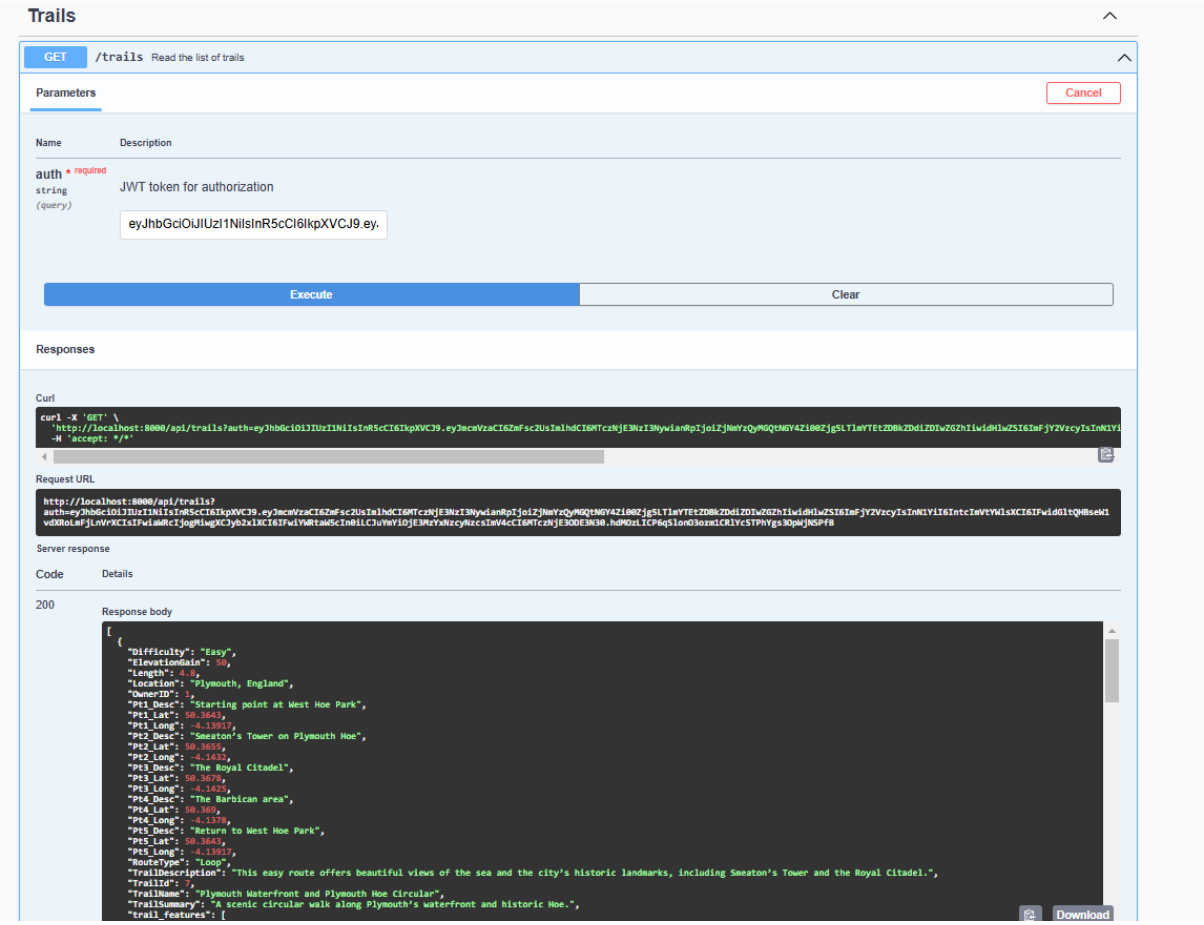
```
(base) C:\Users\Matthew>curl -X POST http://localhost:8000/swagger -H "Content-Type: application/json" -d "{\"email\": \"test\"
, \"password\": \"Test\"}"
{
  "message": "Invalid credentials"
}
```

```
(base) C:\Users\Matthew>curl -X POST http://localhost:8000/swagger -H "Content-Type: application/json" -d "{\"email\": \"tim@pl
ymouth.ac.uk\", \"password\": \"COMP2001!\"}"
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTczNjE3NzI3NywianRpIjoiZjNmYzQyMGQtNGY4Zi0
0Zjg5LTlmYTEtZDBkZDdiZDIwZGZhIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6IntcImVtYWlsXCI6IFwidGltQHBseW1vdXRoLmFjLnVrXCIsIFwiaWRcIjogMiwgXC
Jyb2xlXCI6IFwiYWRtaW5cIn0iLCJuYmYiOjE3MzYxNzcyNzcsImV4cCI6MTczNjE3ODE3N30.hdMOzLICP6qSlonO3ozm1CRlYcSTPhYgs3OpWjNSPf8",
  "redirect_url": "http://localhost:8000/api/ui"
}
```

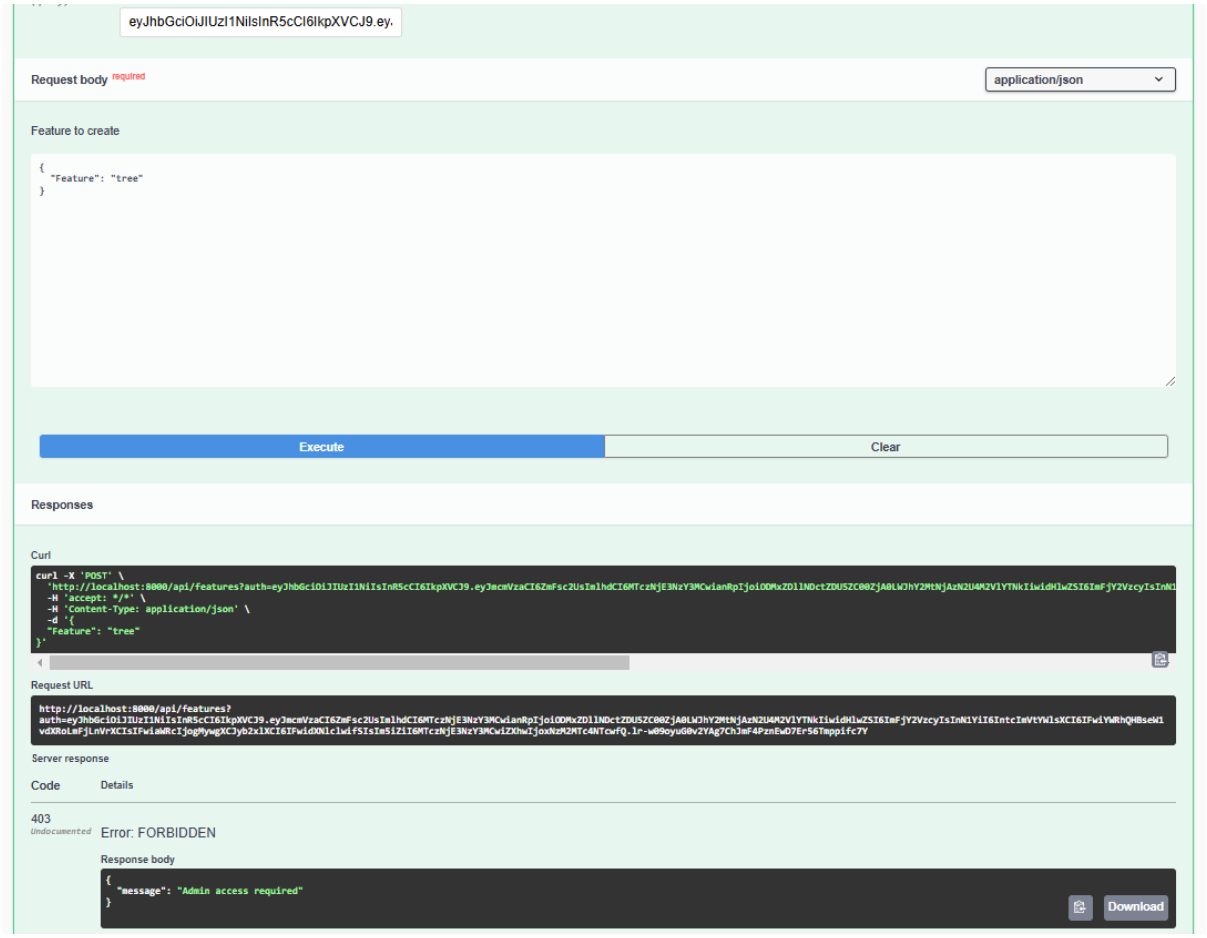
```
(base) C:\Users\Matthew>docker run -p 8000:8000 mattrfish/2001_flask
 * Serving Flask app 'config'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8000
 * Running on http://172.17.0.2:8000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 908-561-063
172.17.0.1 - - [06/Jan/2025 15:27:21] "POST /swagger HTTP/1.1" 403 -
172.17.0.1 - - [06/Jan/2025 15:27:57] "POST /swagger HTTP/1.1" 200 -
```

I have done a POST request to the api, first I have done an incorrect one which gives the error messages. Then I did the correct one which has been verified and I have received the access token.

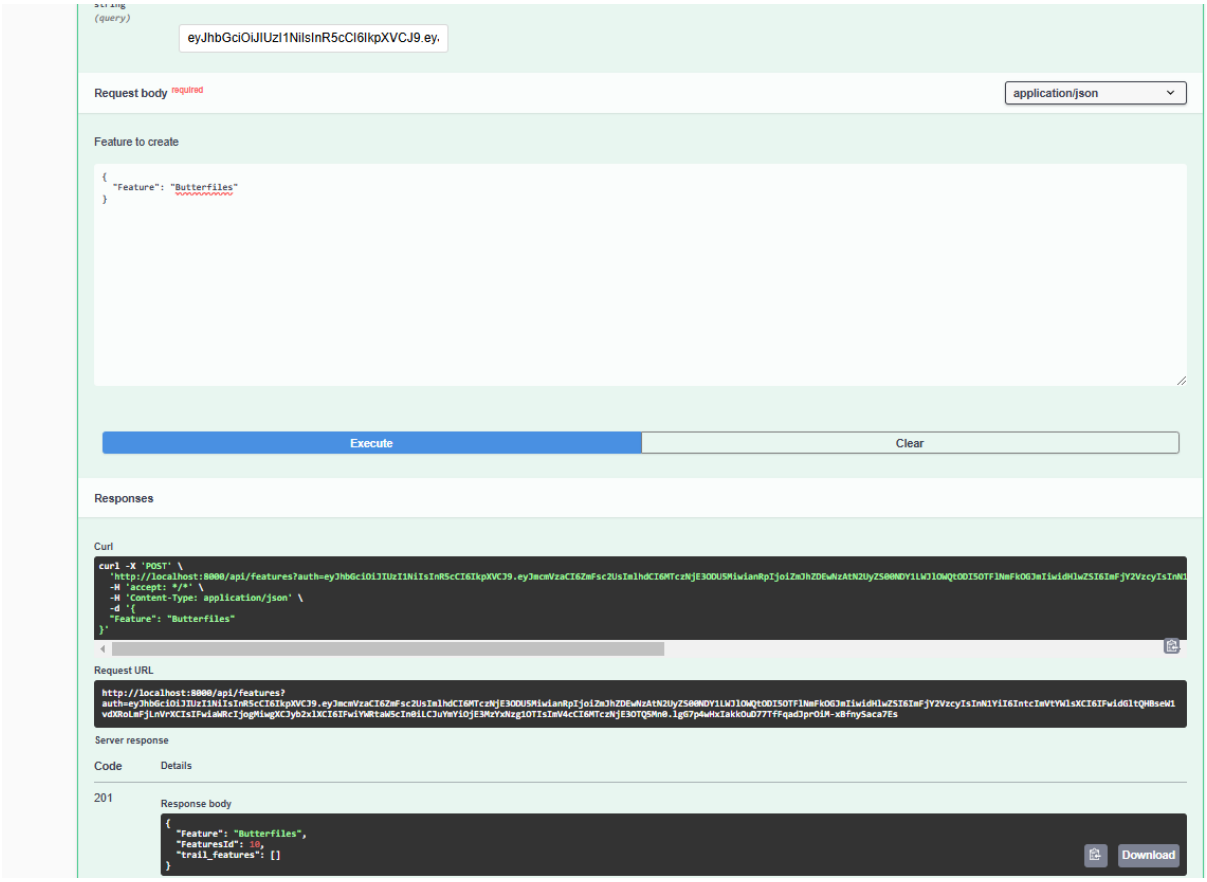
Admin/User Operations



Here I have entered the correct access token and it has given me access.

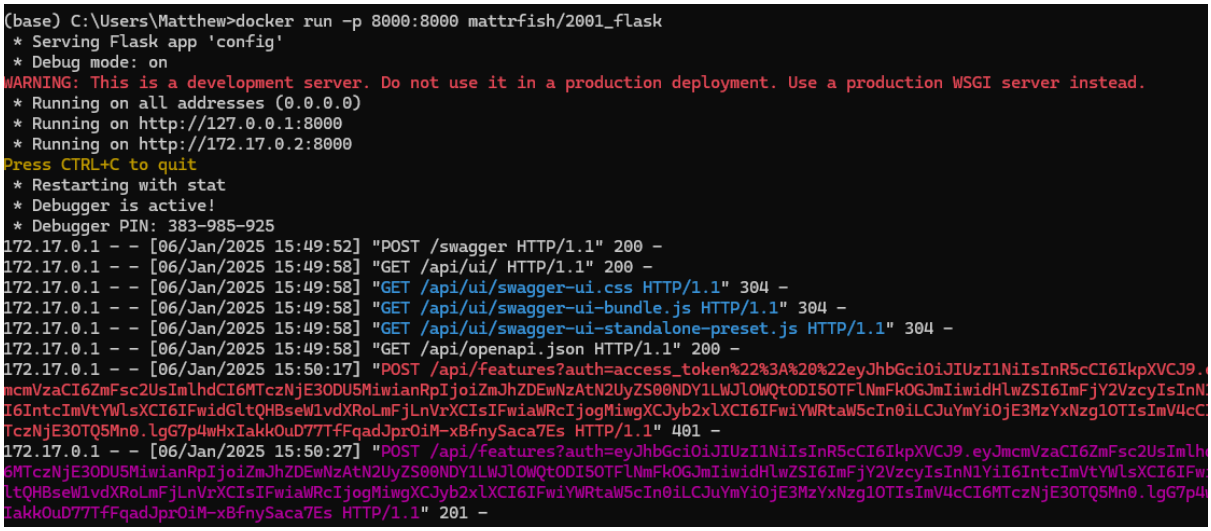


Now I have entered a user access token, which doesn't allow them to add a feature unless they provide the admin access token.



	FeaturesID	Feature
1	1	lake
2	3	Rocky Path
3	4	stream
4	5	pond
5	7	Dog Friendly
6	8	Cafe
7	9	Farm
8	10	Butterfiles
9	*NULL*	*NULL*

I have provided the correct admin auth code and have added a feature to the database.



Areas for Improvement

I have included basic error handling and checking but I think that if I had extra time I would include better security and validation of data to ensure that it is robust.

Although tracking the distance between location points was not in the scope, I think it would have been a good addition to work out how far each points are for the trail, so that users can decide to just walk to a certain point of the trail and not have to do the whole trail.

I have got the user to input their owner id for the trail when creating a trail, but it would have been much better if it could have automatically added the owner id to the trail when the user creates one.

I would also have liked to complete a log table of the owner id, trail made and time it was made just to provide extra security by keeping a log of all the operations being made to the database. This ensures integrity and keeps track of who is doing what.

Conclusion

The trail microservice was successfully designed and implemented, providing core functionalities for managing trail in the trail application. The microservice follows the RESTful principles and uses the authentication API to securely authenticate users to access the microservice. I have ensured that it adheres to legal, social, ethical standards and implements professional best practices in code quality and security. The testing has provided that the service performs as expected, with appropriate handling of user authentication, trail management and data security. Furter improvements could be made to focus on enhancing security measures and adding new features for better user experience.