# Assessment 2: Report

COMP2001 CW2, Corey Richardson

## Introduction

This document will provide a report on the design, development and implementation of a micro-service API to manage the creation, reading, updating and deletion of trails and users, and associated attributes.

## Background

The application is a web-service RESTful API and micro-service that could be used in a larger context to enable users to explore and manage information about walking trails, in a similar manner as is done on *alltrails.com*.

## Assessment Materials

|  |  |
| --- | --- |
| **Docker Container Name** | coreyrichardson1/trails-api |
| **Docker Hub** | <https://hub.docker.com/r/coreyrichardson1/trails-api> |
| **GitHub Repository** | <https://github.com/corey-richardson/comp2001-cw2> |
| **WakaTime Project** | <https://wakatime.com/@coreyrichardson/projects/ouktfmbpqg?start=2024-12-28&end=2025-01-07> |
| **WakaTime Git Commits** | <https://wakatime.com/@coreyrichardson/projects/ouktfmbpqg/commits> |
| **Software Test Document** | <https://github.com/corey-richardson/comp2001-cw2/blob/main/Report/SoftwareTestDocument.pdf> |

The private GitHub repository has been shared with *haoyiwang25*, who has been added as a collaborator. A pending but not yet accepted invite has also been sent to *mjread*.

## Design

## Implementation

The micro-service was implemented using Python’s Flask framework,

|  |
| --- |
| Figure : app.py |

The implementation of the micro-service first involved setting up a Flask application, defining the database models and creating the API’s endpoints and associated methods.

|  |
| --- |
| Figure : config.py |

The connection between the app and the database is configured and established in **config.py**, as shown above. It uses SQLAlchemy modules as the Object-Relational Mapper (ORM) to handle database interactions and Marshmallow for schema serialisation/deserialization and schema validation.

|  |
| --- |
| Figure : ‘Point’ and ‘PointSchema’ Model |

These SQLAlchemy models are representations of database entities, such as a Point. Each model relates to a table in the overall database schema and defines the structure, relationship and constraints of its fields.

The ‘Point’ model represents individual locations that make up a trail. They are composed of the latitude and longitude describing the geographical position of the point, a description that can be used to summarise and describe the point, and optional links to other points in a doubly-linked-list-like structure. This structure design allows trails to be constructed as an ordered sequence of points, allowing for linear and looped trails to be represented of any length without limits.

The ‘Trail’ model stores the ID of a single Point object as ‘starting\_point\_id’ to provide an entry point into the sequence of points. It also includes other attributes such as the trails name, difficulty from a set of allowed values and length. These trails are also optionally associated with a ‘User’ via the ‘author\_id’ foreign key attribute and can be linked to multiple ‘Feature’ objects in a many-to-many relationship through the ‘TrailFeature’ link-entity table.

The Marshmallow defined schemas expand on the SQLAlchemy models and handle the serialization and deserialization of data, converting the models from database-entities to JSON data, compatible for API communication. In reverse, these schemas can transform JSON response payloads back into instances of the model class.

|  |
| --- |
| A computer screen shot of a program  Description automatically generated  Figure 4: SQL Definition of the 'Point' Table |

These schemas can also be used to check the validity of data. For example, the ‘PointSchema’ checks that latitude and longitude values are within valid ranges at the API level, not requiring the database itself to be queried. While these constraints are also enforced at the SQL schema level, it is unlikely to cause issues in the future as these constraints are inherently static and won’t require later modification.

Similar validation is carried out on Trails, as defined in ‘TrailSchema’, ensuring that the ‘route\_type’ field matches defined options, ensuring the integrity of data in the database. This duplication of validation could cause future issues if the list of accepted values is changed in only one location.

|  |
| --- |
| Figure : ‘Point.create()’ |

The methods for creating, updating or deleting entries from the database are protected using the **@require\_auth** decorator, to ensure that only authorised users can access these operations.

|  |
| --- |
| Figure : ‘@require\_auth’ Decorator Method |

The decorator wraps the ‘create’, ‘update’ and ‘delete’ methods of each endpoint and checks the validity of a JSON Web Token (JWT) passed through the “Authorization” header of a HTTP request. If the passed token is missing, invalid or expired then the **validate\_token()** method will abort the request with the HTTP status code *401* indicating *Unauthorized* access. By including this authentication check, the application ensures that only authorized requests can execute these protected database operations, whilst still maintaining read-only access to the database for all other users.

|  |
| --- |
| Figure : 'authenticate()' Method |

The **authenticate** function gets called when a HTTP POST request is sent to the **/login** endpoint of the API and expects and extracts an email and password from the JSON payload of the request body. If present, it will validate these credentials using the University of Plymouth Authenticator API. If the external API confirms that the credentials are valid, the **authenticate** function will generate a JWT. This token will include the user’s email address are part of its payload and is set to expire an hour after it is generated, mitigating the impact of a leaked or stolen token. The JWT token is encrypted using the Symmetric HS256 algorithm to ensure its authenticity and the integrity of requests.

The JWT token is returned to the client to be included in any subsequent requests as part of the “Authorization” header.

As part of the authentication process, the authenticate function also adds the authenticated user to the User table in the database with an “ADMIN” role. This is done by sending the users details as a POST request to the API’s **/user** endpoint. The provided JWT token is automatically included in this request’s header to verify that the operation is authorised.

|  |
| --- |
| Figure : swagger.yml - 'Point' Creation and Read All |

The Swagger file creates the OpenAI specification of the API and documents the endpoints through the **/api/ui** path. It includes metadata describing the micro-services specification, defines schemas and structures of the data being passed through requests and responses, defines paths and endpoints detailing how the API is to be used and describes possible responses for each route.

## Legal, Social, Ethical and Professional Considerations

*One consideration to make moving forward with a future iteration of the project would be the protection of GET/READ methods associated with the* ***/User*** *endpoint. These endpoints currently aren’t protected by the* ***@require\_auth*** *decorator meaning that non-admin users can fetch Personally Identifiable Information (PII) such as user email addresses from the API.*

As of commit **bf05ff3**, the User **GET** endpoints are protected by the **@require\_auth** decorator and so require a valid JWT token in the “Authorization” header of the HTTP request.

## Evaluation

Evidence of the testing process can be found in the **SoftwareTestDocument.pdf** document, found in the **Report/** directory of my GitHub repository or linked to in the table found in the [*Assessment Materials*](#_Assessment_Materials) section of this report. Testing was carried out on a fresh, empty instance of the schema and used Postman to send HTTP requests to the API’s endpoints and verify that responses matched their expected values.

The current iteration of the application only allows for individual creation of Points through POST requests to the **/Point** endpoint, however as I discovered in my testing this can become very tedious when wanting to create a Trail consisting of a large number of Points. The ability to bulk insert Points to the database by passing a JSON list in one request could improve this feature of the micro-service.

Alternatively, this feature could be handled by a front-end to the micro-service, where the user can add Points via a form and the application could iterate over them adding them to the database one at a time. A decision would have to be made here on whether insertions should be validated individually or as a batch, so that if any of the points are not valid then none of the points would be committed into the database.