

# 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.

The **Software Test Document** linked in the below *Assessment Materials* table showcases the testing and validation checks conducted on the micro-service.

## 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*.

The system implements a micro-service architecture and is therefore modular, allowing it to be extending or integrated into other services in a larger context with ease. The use of a RESTful API means that the service can be easily accessed through HTTP requests, allowing it to easily interact with other services.

## Assessment Materials

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

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

### Entity-Relationship Diagram (ERD)

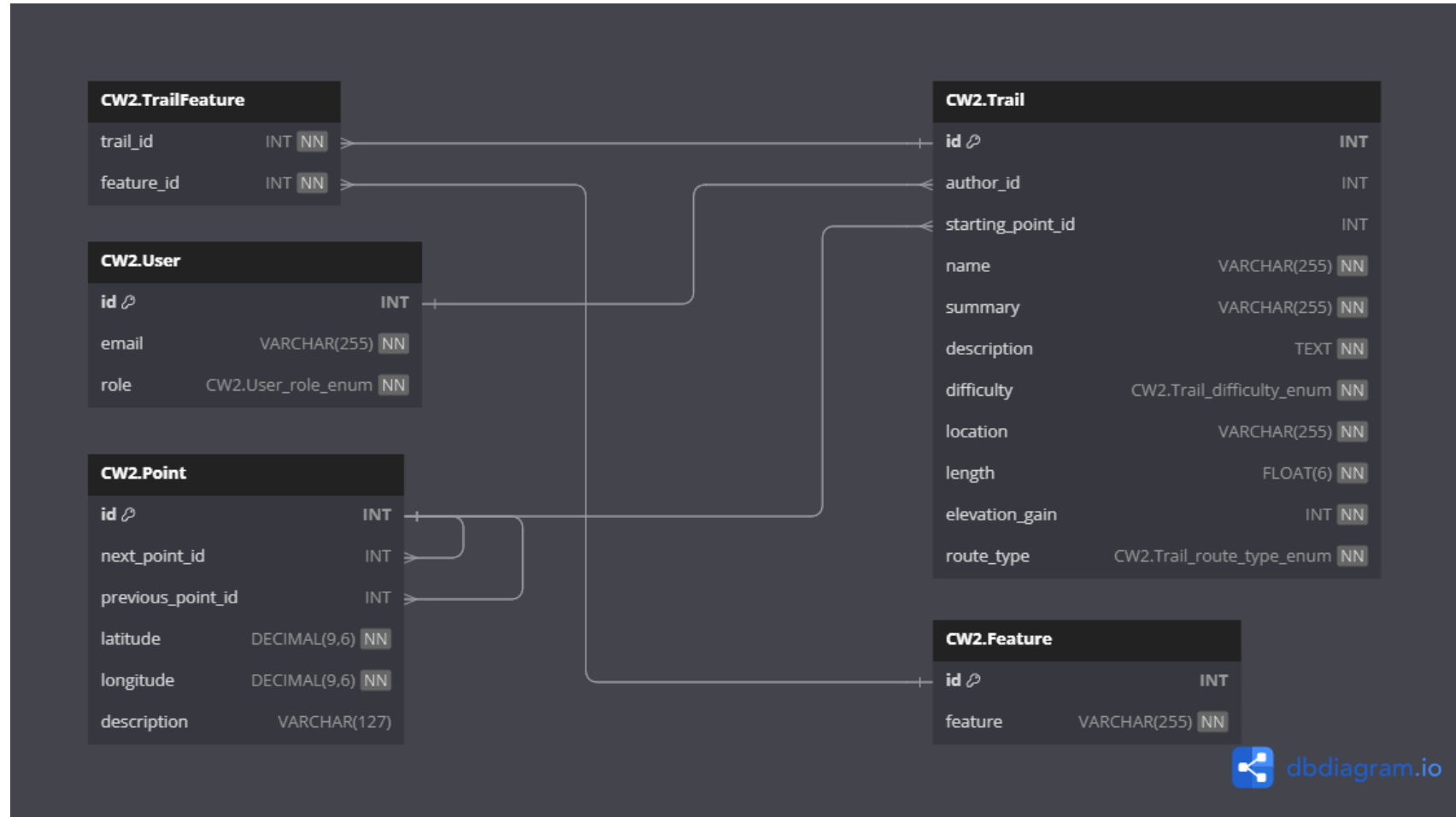


Figure 1: Entity-Relationship Diagram

## Class Diagrams

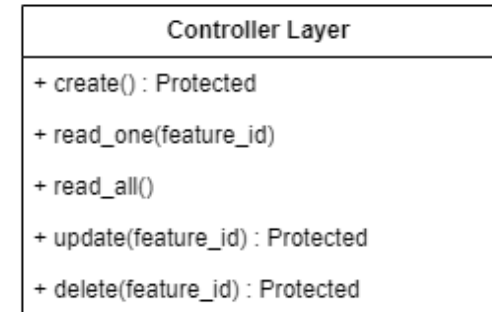
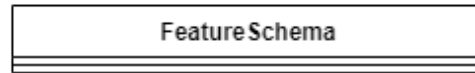
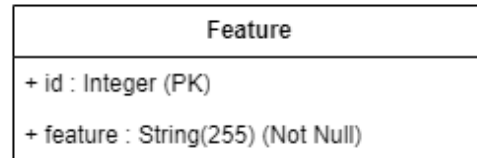


Figure 2: Feature and FeatureSchema Class Diagrams

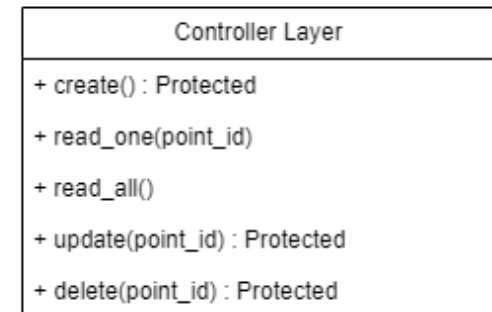
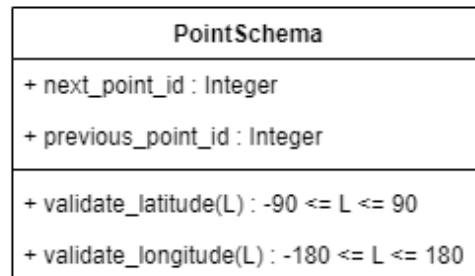
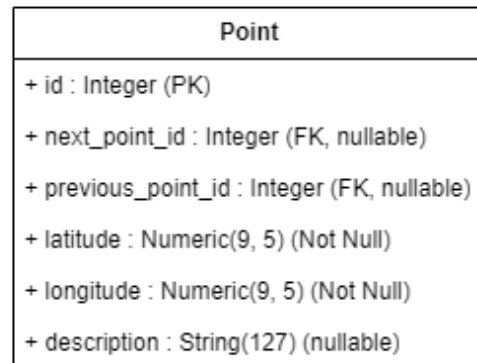


Figure 3: Point and PointSchema Class Diagrams

Trail
+ id : Integer (PK)
+ author_id : Integer (FK, nullable)
+ starting_point_id : Integer (FK, nullable)
+ name : String(255) (Not Null)
+ summary : String(255) (Not Null)
+ description : Text (Not Null)
+ difficulty : String(9) (Not Null)
+ location : String(255) (Not Null)
+ length : Float (Not Null)
+ elevation_gain : Integer (Not Null)
+ route_type : String(15) (Not Null)

TrailSchema
+ author_id : Integer
+ starting_point_id : Integer
+ validate_difficulty(value) : Enum
+ validate_route_type(value) : Enum

Controller Layer
+ create() : Protected
+ read_one(trail_id)
+ read_all()
+ update(trail_id) : Protected
+ delete(trail_id) : Protected

Figure 4: Trail and TrailSchema Class Diagrams

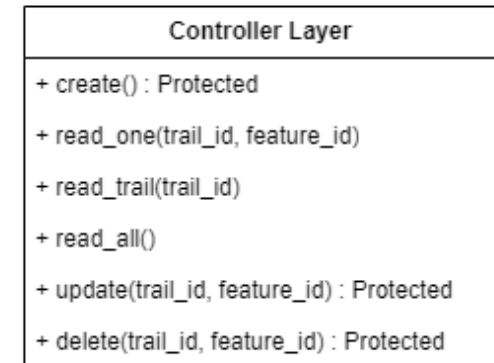
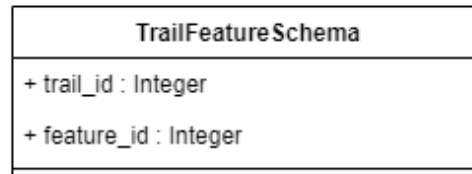
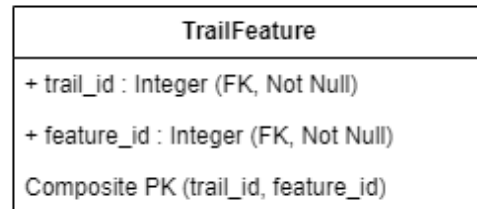


Figure 5: TrailFeature and TrailFeatureSchema Class Diagrams

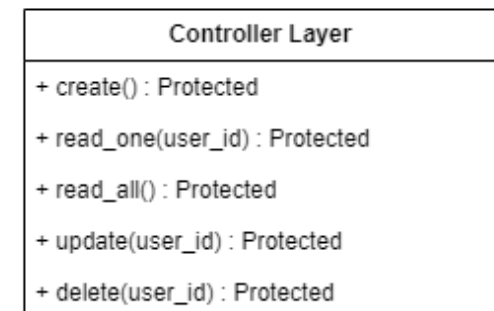
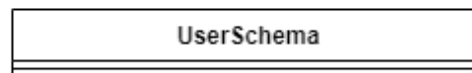
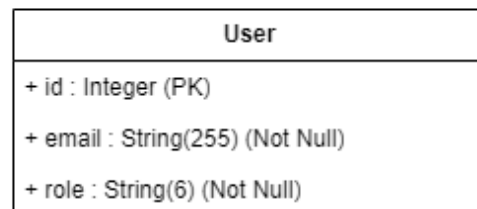
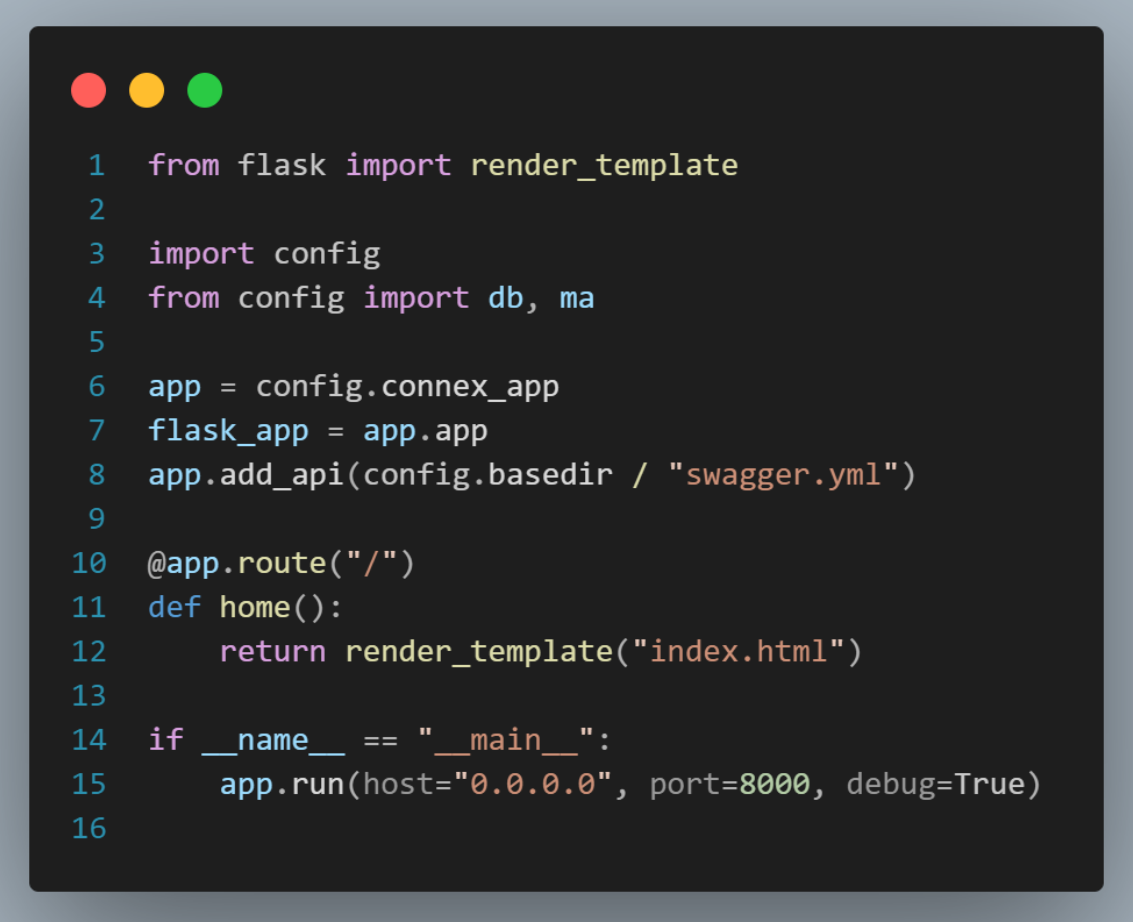


Figure 6: User and UserSchema Class Diagrams

## Implementation


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



```
1  from flask import render_template
2
3  import config
4  from config import db, ma
5
6  app = config.connex_app
7  flask_app = app.app
8  app.add_api(config.basedir / "swagger.yml")
9
10 @app.route("/")
11 def home():
12     return render_template("index.html")
13
14 if __name__ == "__main__":
15     app.run(host="0.0.0.0", port=8000, debug=True)
16
```

Figure 7: *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.



```

1  import pathlib
2  import connexion
3  from flask_sqlalchemy import SQLAlchemy
4  from flask_marshmallow import Marshmallow
5
6  basedir = pathlib.Path(__file__).parent.resolve()
7  connex_app = connexion.App(__name__, specification_dir=basedir)
8
9  # Database credentials
10 server = "DIST-6-505.uopnet.plymouth.ac.uk"
11 database = 'COMP2001_CRichardson'
12 username = "CRichardson"
13 password = "BssN103*"
14
15 app = connex_app.app
16 app.config["SQLALCHEMY_DATABASE_URI"] = (
17     "mssql+pyodbc:///?odbc_connect="
18     "DRIVER={ODBC Driver 17 for SQL Server};"
19     f"SERVER={server};"
20     f"DATABASE={database};"
21     f"UID={username};"
22     f"PWD={password};"
23     "TrustServerCertificate=yes;"
24     "Encrypt=yes;"
25 )
26
27 app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
28
29 # Initialise SQLAlchemy and Marshmallow in the context of the app
30 db = SQLAlchemy(app)
31 ma = Marshmallow(app)
32

```

Figure 8: 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.

```

1  from config import db, ma
2
3  from marshmallow_sqlalchemy import fields
4  from marshmallow import fields, validates
5
6  class Point(db.Model):
7      """Model for a point in a trail."""
8      __tablename__ = "CW2.Point"
9
10     id = db.Column(db.Integer, primary_key = True, autoincrement = True)
11     next_point_id = db.Column(db.Integer, db.ForeignKey("CW2.Point.id"), nullable = True)
12     previous_point_id = db.Column(db.Integer, db.ForeignKey("CW2.Point.id"), nullable = True)
13     latitude = db.Column(db.Numeric(9, 6), nullable = False)
14     longitude = db.Column(db.Numeric(9, 6), nullable = False)
15     description = db.Column(db.String(127), nullable = True)
16
17     next_point = db.relationship(
18         "Point", remote_side=[id], foreign_keys=[next_point_id], backref="previous_points"
19     )
20     previous_point = db.relationship(
21         "Point", remote_side=[id], foreign_keys=[previous_point_id], backref="next_points"
22     )
23
24
25     class PointSchema(ma.SQLAlchemyAutoSchema):
26         """Schema for de/serialising a Point"""
27         class Meta:
28             model = Point
29             load_instance = True
30             sqla_session = db.session
31
32         next_point_id = fields.Integer()
33         previous_point_id = fields.Integer()
34
35         # https://docs.sqlalchemy.org/en/20/orm/mapped_attributes.html
36         @validates("latitude")
37         def validate_latitude(self, value):
38             if not -90 <= value <= 90:
39                 raise ValueError(f"Failed Latitude Validation. Failed check: -90 <= {value} <= 90")
40
41         @validates("longitude")
42         def validate_longitude(self, value):
43             if not -180 <= value <= 180:
44                 raise ValueError(f"Failed Longitude Validation. Failed check: -180 <= {value} <= 180")
45

```

Figure 9: '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.

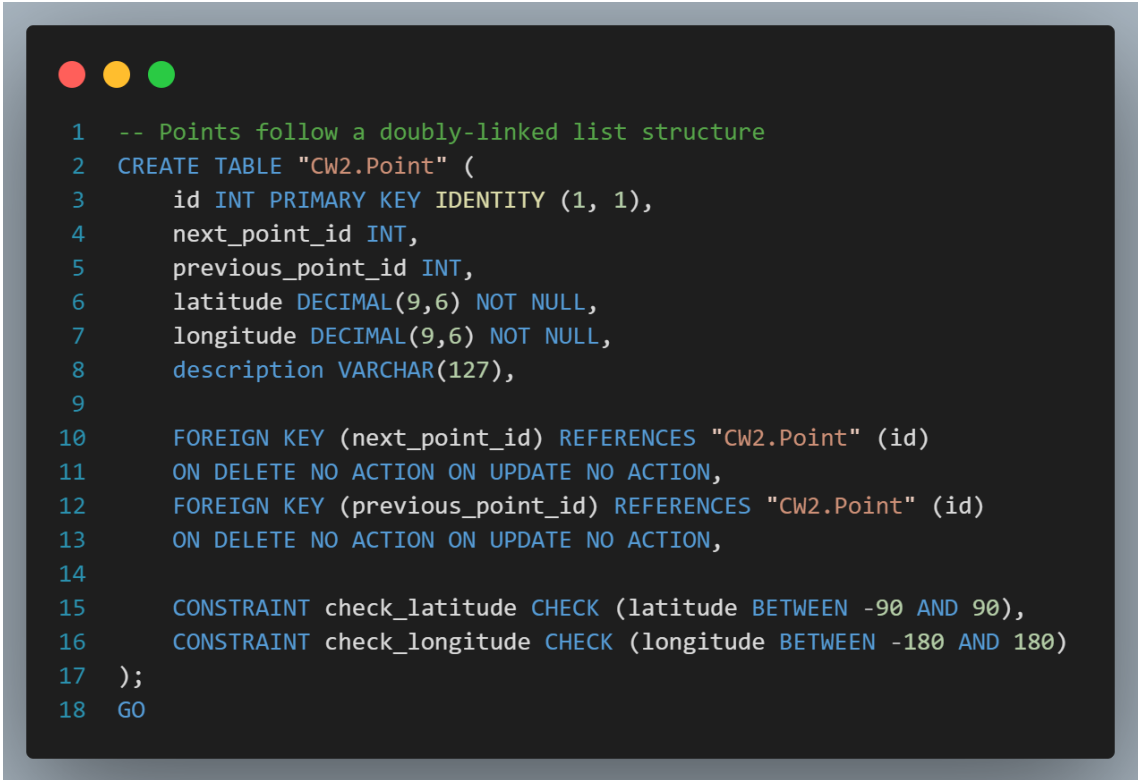
The 'Point' model represents individual locations that make up a trail. They are composed of the latitude and longitude describing the geographical position of the point, a description that can be used to summarise and describe the point, and optional links to other points in a doubly-linked-list-like 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.

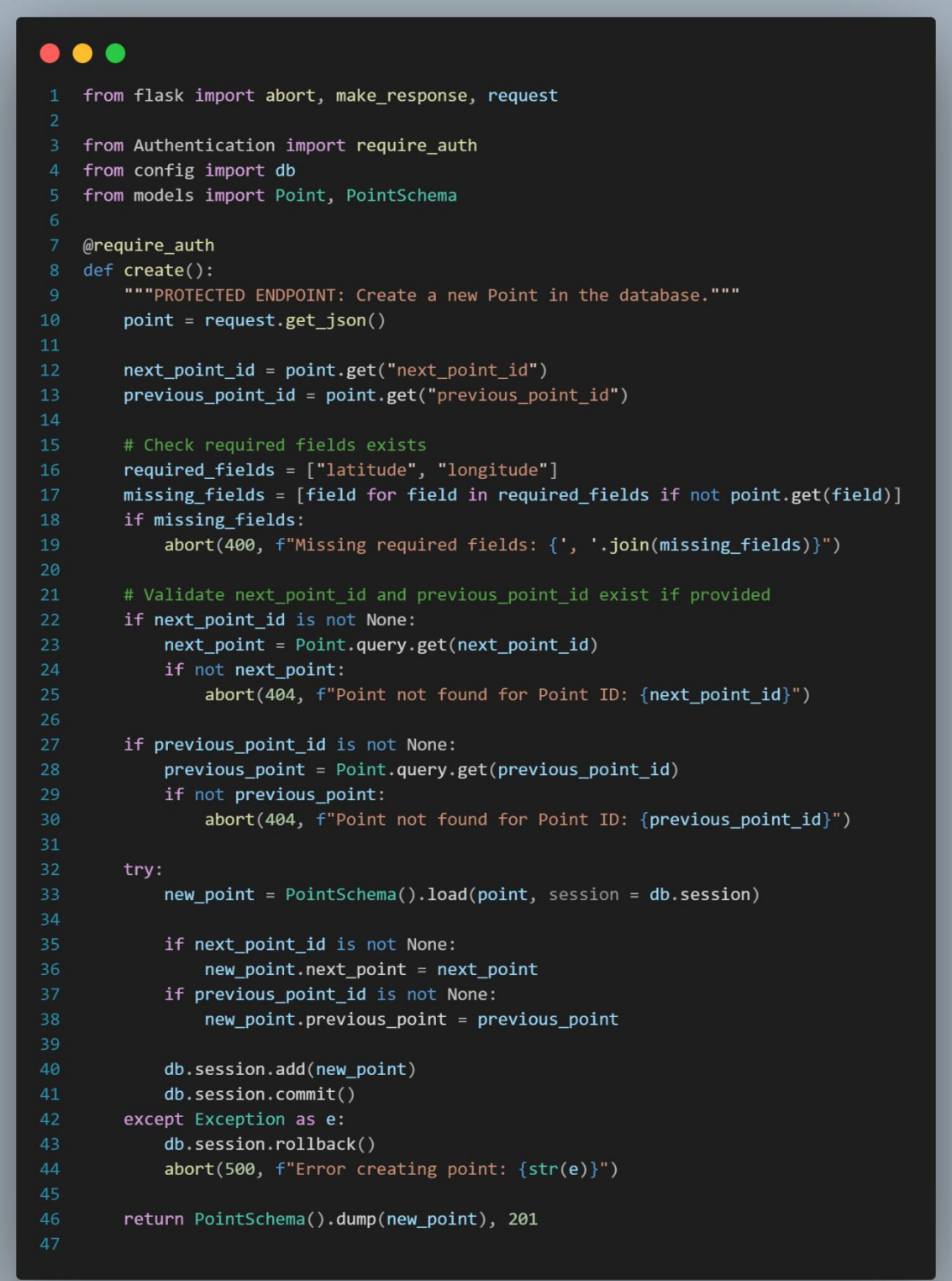


```
1  -- Points follow a doubly-linked list structure
2  CREATE TABLE "CW2.Point" (
3      id INT PRIMARY KEY IDENTITY (1, 1),
4      next_point_id INT,
5      previous_point_id INT,
6      latitude DECIMAL(9,6) NOT NULL,
7      longitude DECIMAL(9,6) NOT NULL,
8      description VARCHAR(127),
9
10     FOREIGN KEY (next_point_id) REFERENCES "CW2.Point" (id)
11     ON DELETE NO ACTION ON UPDATE NO ACTION,
12     FOREIGN KEY (previous_point_id) REFERENCES "CW2.Point" (id)
13     ON DELETE NO ACTION ON UPDATE NO ACTION,
14
15     CONSTRAINT check_latitude CHECK (latitude BETWEEN -90 AND 90),
16     CONSTRAINT check_longitude CHECK (longitude BETWEEN -180 AND 180)
17 );
18 GO
```

Figure 10: 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.



```

1  from flask import abort, make_response, request
2
3  from Authentication import require_auth
4  from config import db
5  from models import Point, PointSchema
6
7  @require_auth
8  def create():
9      """PROTECTED ENDPOINT: Create a new Point in the database."""
10     point = request.get_json()
11
12     next_point_id = point.get("next_point_id")
13     previous_point_id = point.get("previous_point_id")
14
15     # Check required fields exists
16     required_fields = ["latitude", "longitude"]
17     missing_fields = [field for field in required_fields if not point.get(field)]
18     if missing_fields:
19         abort(400, f"Missing required fields: {'', '.join(missing_fields)}")
20
21     # Validate next_point_id and previous_point_id exist if provided
22     if next_point_id is not None:
23         next_point = Point.query.get(next_point_id)
24         if not next_point:
25             abort(404, f"Point not found for Point ID: {next_point_id}")
26
27     if previous_point_id is not None:
28         previous_point = Point.query.get(previous_point_id)
29         if not previous_point:
30             abort(404, f"Point not found for Point ID: {previous_point_id}")
31
32     try:
33         new_point = PointSchema().load(point, session = db.session)
34
35         if next_point_id is not None:
36             new_point.next_point = next_point
37         if previous_point_id is not None:
38             new_point.previous_point = previous_point
39
40         db.session.add(new_point)
41         db.session.commit()
42     except Exception as e:
43         db.session.rollback()
44         abort(500, f"Error creating point: {str(e)}")
45
46     return PointSchema().dump(new_point), 201
47

```

Figure 11: '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.



```

1  def validate_token():
2      """
3      Decodes and validates a JWT token from the
4      "Authorization" header of the request.
5      """
6      auth_header = request.headers.get("Authorization")
7
8      if not auth_header:
9          abort(401, "Missing token.")
10     if not auth_header.startswith("Bearer "):
11         abort(401, "Invalid token format.")
12
13     # Parse the token
14     token = auth_header.split(" ")[1].strip()
15
16     try:
17         # Decode
18         jwt.decode(token, SECRET_KEY, algorithms = ["HS256"])
19     except jwt.ExpiredSignatureError:
20         abort(401, "Token expired")
21     except jwt.InvalidTokenError:
22         abort(401, "Invalid token")
23
24
25 def require_auth(f):
26     """
27     Decorator to mark endpoint methods that should be protected.
28     """
29     # print(f.__name__)
30     @wraps(f)
31     def decorator(*args, **kwargs):
32         validate_token()
33         return f(*args, **kwargs)
34     return decorator

```

Figure 12: '@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.

```

1 AUTH_URL = "https://web.socem.plymouth.ac.uk/COMP2001/auth/api/users"
2 USER_URL = "http://localhost:8000/api/user"
3
4 SECRET_KEY = "SECRET_KEY_DONT_LOOK_AT_ME"
5
6 def authenticate():
7     """
8     Handles user authentication by validating an email and password with
9     the University of Plymouth Authenticator API.
10    """
11    # Retrieve request body and check validity
12    body = request.get_json()
13    if not body or not body.get("email") or not body.get("password"):
14        abort(400, "Email and password fields are required.")
15
16    # Create request body JSON
17    credentials = {
18        "email" : body["email"],
19        "password" : body["password"]
20    }
21
22    try:
23        # POST request to Authenticator API
24        response = requests.post(AUTH_URL, json=credentials)
25    except requests.exceptions.RequestException as e:
26        abort(500, f"Failed to authenticate: {str(e)}")
27
28    # Was authentication successful?
29    if response.status_code == 200:
30        try:
31            json_response = response.json()
32
33            if json_response[1] == "True":
34                # Generate a JWT Token
35                expiration = datetime.now(timezone.utc) + timedelta(hours=1)
36                payload = {
37                    "email" : body["email"],
38                    "exp" : expiration
39                }
40
41                token = jwt.encode(payload, SECRET_KEY, algorithm = "HS256")
42
43            except requests.JSONDecodeError:
44                abort(500, f"Error processing authentication response: {str(e)}")
45
46            # Add user to the server database with an ADMIN role
47            user_data = {
48                "email" : body["email"],
49                "role" : "ADMIN"
50            }
51
52            auth_header = {
53                "Authorization" : "Bearer " + token
54            }
55
56            try:
57                response = requests.post(USER_URL, json=user_data, headers=auth_header)
58            except Exception as e:
59                print("Failed to add new user: ", str(e))
60
61        else:
62            abort(401, "Invalid credentials")
63
64    return {"token" : token}, 200
65

```

Figure 13: '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 as 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 user's 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.

```

1  /point:
2    post:
3      operationId: "Point.create"
4      tags:
5        - "Point"
6      summary: "Create a point"
7      requestBody:
8        description: "Point to create"
9        required: true
10       content:
11         application/json:
12           schema:
13             $ref: "#/components/schemas/Point"
14       responses:
15         "201":
16           description: "Point created successfully"
17           content:
18             application/json:
19               schema:
20                 $ref: "#/components/schemas/Point"
21         "400":
22           description: "Missing required fields"
23         "401":
24           description: "Missing, invalid or expired authentication token"
25         "404":
26           description: "Point not found (Foreign Key Constraint)"
27         "500":
28           description: "Error creating point"
29       get:
30         operationId: "Point.read_all"
31         tags:
32           - "Point"
33         summary: "Read the list of points"
34         responses:
35           "200":
36             description: "Successfully read point list"
37

```

Figure 14: swagger.yml - 'Point' Creation and Read All

The Swagger file creates the OpenAPI 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.*

This was addressed by commit **bf05ff3**, where the User **GET** endpoints are now protected by the **@require\_auth** decorator and therefore these routes now require a valid JWT token in the "Authorization" header of the HTTP request, preventing unauthorized actors from accessing PII.

As the API manages user data, including PII, it should comply with data protection regulations such as the Data Protection Act 2018.

The API uses the symmetric HS256 algorithm to encrypt the JWT security tokens to reduce the risk of token tampering, however no encryption is carried out on stored user data, at rest or during transit. This could be addressed by future iterations of the application.

The database stores the minimum required PII in line with General Data Protection Regulation (GDPR) guidelines, mitigating the impact of any potential data breach; only the user's email address is stored in the server's database as password authentication is handled by an external service.

While the current application is primarily a backend service, a hypothetical frontend user interface developed in a future iteration would be required to comply with Web Content Accessibility Guidelines (WCAG) to ensure that the application is accessible to all users, including users with disabilities. This would be a social and ethical consideration of the application.

Additionally, a feature to prevent API abuse would be beneficial to a future iteration of the design to protect the service and the data stored in relation to users. This could be implemented through features such as rate limiting, suspicious activity logging to improve the auditing processes and an improved Authorization methodology.

## 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](#) section of this report. Testing was carried out on a fresh, empty instance of the schema and used Postman as a tool to send HTTP requests to the API's Create, Read, Update and Delete endpoints and verify that actual responses matched the expected responses, and returned the expected payloads. Testing including valid and invalid request bodies, edge cases and authentication scenarios.

As well as testing endpoints in individual scenarios, the workflow for creating a new trail was also tested. This included the full process such as the creation and linking of multiple geographical Points, the creation of a Trail and the creation and linking of a Feature using the TrailFeature endpoint and table.

During the test process, it was also validated that JWT tokens expire after an hour, as proven by a **401 Unauthorized** response midway through the test process.

Error handling worked as expected. When users sent invalid data, the API returned the appropriate HTTP response status code, such as *400 Bad Request* when the request body didn't contain all required parameters, providing an informative debugging tool for users of the API to understand and correct the issues with their requests.

I also utilised Postman's *Collection* feature to create a Test Sequence which will run through a predefined series of HTTP requests and compare the results against a set response status code. This allows for faster testing improving the efficiency at which new features could be added. This test Collection can be seen in my GitHub repository at [/Testing/TestSequence.postman\\_collection.json](#) and the exported results can be viewed at [/Testing/TestSequence.postman\\_test\\_run.json](#).

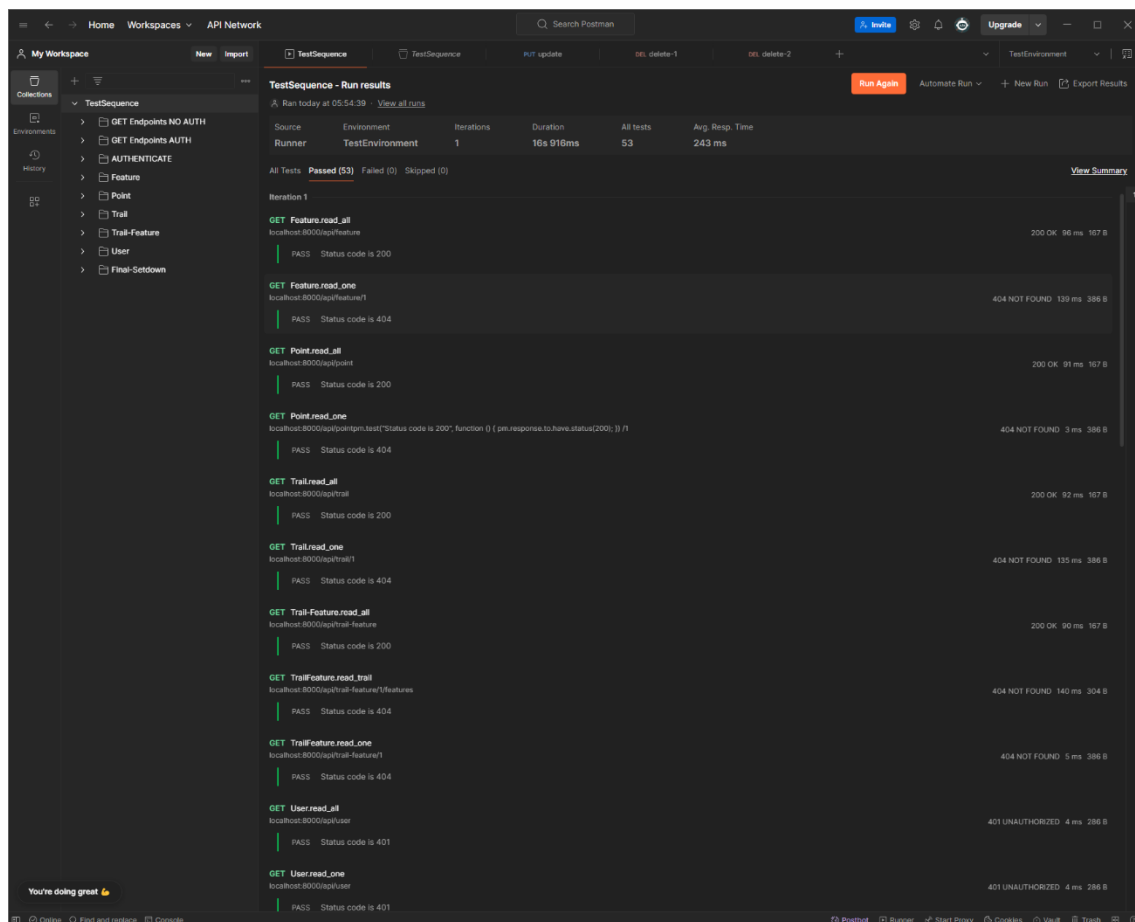


Figure 15: Results of running a Postman Test Collection

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.



As discussed in the Implementation section of my report, there is some duplication of validation logic between the database schema and Marshmallow module schemas which could cause issues if the list of accepted values is updated in one location but not in another. One section of the codebase could be enforcing outdated logic. To improve this, the validation logic should be moved to be centralised so that changes only need to be reflected in one location.

More endpoints allowing improved query filtering could be useful as the service scales, such as allowing the users of the micro-service to filter trails based on their attributes, such a difficulty, length or features.