Django Trail API MicroService

GITHUB: **https://github.com/alfie-ns/2001-cw**

DOCKER: oladeanio/trail-api {

1- `docker pull oladeanio/trail-api:latest`

2- `docker run -it -p 8000:8000--nametrail-api-container oladeanio/trail-api:latest`

---

to kill docker: `docker stop $(docker ps -aq) && docker rm $(docker ps -aq)`

}

Introduction

This report documents the design and implementation of my *Django Trail API micro-service*. The service is a RESTful API that facilitates the management of trails and its respective data; this report also addresses by referencing how this solution meets the specified learning outcomes {

1- Demonstrate explicit uses of modelling techniques to gain access to information and data to support a given need.

2- Illustrate an appropriate technical solution to the problems of information privacy, integrity, security and preservation.

3- Illustrate the design of an application of moderate complexity to elicit and visualise information from a data store.

} (1-3)

To use my **Django admin account**

sign in with { username: `oladeanio` password: `root` }

In this page, the user i.e. me, can add, edit or delete trails in the database

API documentation is generated by swagger; access it with
`http://127.0.0.1:8000/swagger`

Background

The Trail API micro-service is a **Django** application that provides a **RESTful API** for managing trails. The service is designed to be used by a front-end template(**manage.html**) to manage trails, including creating, updating, and deleting trails. The API also provides endpoints for retrieving trails and trail details; specifically: {

- CRUD operations for trail management (1, 3)
- Location-based storage and retrieval of trails (1)
- User ownership and access control (2)
- Integration with an existing authentication service (2)

}

Design

The design incorporates key elements such as models, serialisers, and views to ensure effective data management and visualisation. UML diagrams were used to illustrate the relationships between entities, aligning with modelling techniques (1).

Implementation

Project Setup and Dependencies

The Trail API micro-service is containerised using a Dockerfile to ensure portability and consistent deployment across environments. The project dependencies, such as Django, Django REST Framework (DRF), drf-yasg for Swagger documentation, and mssql-django for SQL Server integration, are listed in a requirements.txt. This setup allows for easy installation(`pip install -r requirements.txt`) and environment replication, thus improving the development and deployment workflow (3).

Models

The core data structure of the API is represented by two models: `Trail` and `Location`. The `Trail` model captures essential trail attributes such as `name`, `length`, `difficulty`, `route_type`, and its association with a `Location` and `User`. The `Location` model stores geographical data, including `city`, `county`, and `country`. These models align with the database schema and are implemented using Django's ORM for seamless integration with the SQL Server backend. The foreign key relationships between `Trail` and `Location` models ensure data integrity and consistency (1, 2).

Serialisers

Serialisers play an important role in the Trail API by converting model instances into JSON format for RESTful API responses and validating incoming data. The `TrailSerializer` includes a nested `LocationSerializer`, which ensures that trail and location data are managed together effectively. Custom `create` and `update` methods in the `TrailSerializer` handle nested relationships, providing robust data management capabilities (1, 3).

I've made Django APIs a lot, and i'm pretty sure it needs to be spelt American serializers in the code to work properly.

Views

The API uses Django's `ModelViewSet` to handle CRUD operations for trails and locations. The `TrailViewSet` and `LocationViewSet` reduce boilerplate code by leveraging Django REST Framework's built-in functionality. Custom logic in the `TrailViewSet`, such as assigning a default user in `perform_create`, adds flexibility and ensures proper ownership assignment for newly created trails (2, 3).

Endpoints

The micro-service exposes endpoints for managing trails and locations:

- `/api/trails/manage/`: Provides a front-end template for managing trails (3).
- `/api/trails/`: Handles trail creation, retrieval, updating, and deletion (1, 3).
- `/api/locations/`: Manages location data associated with trails (1).
- `/swagger/`: Provides documentation for exploring and testing the API (3).

Swagger documentation is available at `/swagger/`, providing an interactive interface for exploring and testing the API. These endpoints adhere to RESTful principles, using meaningful URLs and appropriate HTTP methods for resource management (1, 3).

Authentication and Permissions

The API integrates with Django's `User` model to manage trail ownership. By default, new trails are assigned to a `default_user` when authentication is not required. This approach ensures seamless functionality while maintaining flexibility for future authentication enhancements (2).

Deployment

The API is hosted on the university web server(***web.socem.plymouth.ac.uk***) with SQL Server as the database backend(***dist-6-505.uopnet.plymouth.ac.uk***). The use of Docker ensures a consistent deployment environment, while Django's framework provides scalability and reliability for the service (2, 3).

For the Django unit tests, I needed to modify the settings to use a local SQLite database when running tests, since I didn't have sufficient permissions to create test databases on the university's SQL Server. This worked well because SQLite is lighter and doesn't need special permissions, while still letting me properly test that my API works.

The tests check that:

- Creating new trails works (checking POST requests return 201)
- Getting trail data works (checking GET requests return 200)
- The data saved and retrieved matches what was sent

The tests all passed successfully, confirming my API endpoints are working as intended. This gives me confidence that the core CRUD operations of my Trail API are functioning correctly.