

GP Stats: A MotoGP RESTful Data Service (MRDS)

Advanced Web Development midterm project

Student number: ~~8888888885~~

View the project online on:

<http://100.27.228.224:8000/>

Instructions:

Installing and running the project

1. Unzip the file
2. Activate your virtual environment
3. In the console run `cd .\GP_stats\`
4. Run `pip install -r requirements.txt`
5. Run `cd .\GP_stats\`
6. Run `python manage.py makemigrations` + `python manage.py migrate`
7. Run `python Scripts/populate_mrds.py`
8. Run `python manage.py runserver`

A list of all packages and the versions used for your implementation:

- asgiref==3.8.1
- attrs==24.3.0
- Django==5.1.4
- djangorestframework==3.15.2
- drf-yasg==1.21.8
- inflection==0.5.1
- jsonschema==4.23.0
- jsonschema-specifications==2024.10.1
- packaging==24.2
- pytz==2024.2
- PyYAML==6.0.2
- referencing==0.35.1
- rpds-py==0.22.3
- sqlparse==0.5.3
- tzdata==2024.2
- uritemplate==4.1.1

Development environment i.e. the operating system and python version

- Windows 11
- Python 3.11.0

Instruction for logging into the Django-admin site i.e. username and password

1. Go to <http://localhost:8000/admin>
2. Fill in the credentials
 - a. Username: admin
 - b. Password: Gokapusik^32

Include how to run the unit tests and the location of the data-loading script

Before running the server

1. Go to the root folder of the application (GP_stats) where the manage.py file is.
2. In the console type `python manage.py test`

Introduction

This report details the development and implementation of a RESTful data service for MotoGP racing statistics using Django and Django REST Framework (DRF). The application provides

both a web interface and API endpoints for accessing and managing motorcycle racing data across different racing categories (MotoGP, Moto2, Moto3).

The application serves as a MotoGP database, where users can add, remove, edit or simply view the race or rider's info. The endpoints allow for the most crucial data operations that such a portal should provide.

Dataset:

I've chosen this dataset for a few reasons. One is that I'm a fan of MotoGP, but more importantly, I saw a big potential for this project. There is a lot of repetition in the dataset and it is not normalized at all.

<https://www.kaggle.com/datasets/amalsalilan/motogpresultdataset>

This dataset contains comprehensive results of MotoGP races from various years, categories and circuits. It includes information such as the year of the race, the category of the race, the circuit name, the rider and team information, the bike used, the rider's position, the points earned and the country of the rider. The dataset also includes other additional information like the speed, number and time of the rider. This dataset is suitable for anyone looking to analyze the results of MotoGP races and gain insights into the performance of riders, teams and circuits.

The dataset contains 56396 rows and 15 columns and is in a pandas Dataframe format
year: The year in which the race took place.

category: The category of the race such as MotoGP, Moto2, Moto3 etc.

sequence: The sequence of the race in the season.

shortname: A short name of the circuit where the race took place.

circuit_name: The name of the circuit where the race took place.

rider: A unique identifier for the rider.

rider_name: The name of the rider.

team_name: The name of the team the rider was riding for.

bike_name: The name of the bike used by the rider.

position: The final position of the rider in the race.

points: The number of points earned by the rider in the race.

number: The number worn by the rider during the race.

country: The country of the rider.

speed: The average speed of the rider during the race.

time: The time taken by the rider to complete the race

I've removed quite a lot of data from the top and left the records starting from 2016 in order to fit into the 10,000 records limit specified in the instructions for this midterm project.

Normalizing the dataset:

The MotoGP dataset is composed of complete results from several different racing disciplines. The raw dataset is just a single CSV file that contains the headers of year, category, sequence, shortname, circuit_name, rider, rider_name, team_name, bike_name, position, points, number, country, speed, and time. This raw data format had some standardization challenges to overcome.

This was how normalization occurred to produce the Third Normal Form (3NF):

First Normal Form (1NF) was obtained by ensuring that every column had atomic values and no repeating groups. Every row in the original dataset had already fulfilled this condition because it was just a single race result without composite or multi-valued elements.

To achieve **Second Normal Form (2NF)**, I delineated partial dependencies. The circuit (shortname, circuit_name) relied on the circuit identity, not the race outcome as a whole. Likewise, rider information (rider_name, number, country) relied solely on the identity of the rider. This resulted in distinct Circuit and Rider models.

This **Third Normal Form (3NF)** was obtained by eliminating transitive dependencies. I concluded that team_name and bike_name depended on who the team was and not just the race outcome. This prompted the development of a separate Team model. Moreover, the race characteristics (year, category, order) were modeled into a Race model to break relationships between them and the result properties.

This normalisation produced five models:

- Circuit Model - contains circuit data (id, shortname, full_name) representing specific racing locations.
- Rider Model - contains rider data (id, name, country, number) that means you won't see the same information in different race results.
- Team Model - takes care of the team details (id, name, bike_name), thus maintaining team and manufacturer data in the same way in the entire dataset.
- Race Model - stores race information (id, year, category, sequence, circuit_id) which maintains an event's hierarchy.
- Result Model - is the hub that bridges all other models and stores result-relevant data (id, race_id, rider_id, team_id, position, points, speed, time).

API Endpoints and Query Complexity

The app provides seven different API endpoints, which demonstrate a different degree of data manipulation and analysis complexity.

Add Rider Endpoint

The `add_rider` method queries the database and inserts a new rider record. It checks the data passed to it via a serializer and then copies the record with the help of Django ORM, thereby maintaining data integrity. The endpoint takes rider name, number, and country. It saves the rider if the data is correct and returns their ID. Otherwise, it returns validation errors so it is simple and efficient to create new entries.

Add Race Result Endpoint

The `add_race_result` method creates a new race result for a given race. Firstly, it validates the race ID using `get_object_or_404` to make sure that the race passed is valid. It then validates and serializes the resulting data using a serializer. This conclusion also illustrates how nested connections between models can be handled effectively while preserving data consistency by associating results with races.

Top Riders by Category and Year Endpoint

The `top_riders_by_category_and_year` endpoint aggregates deep data. It sorts results by category and year, groups them by rider, and calculates the total points from Django's ORM. This query illustrates efficient data processing on the database level. You can use it to rank riders by their performance. This endpoint could be used by the leaderboards..

Edit Race Result Endpoint

The `edit_race_result` endpoint provides patch updates through PATCH calls. It has the peculiarity of using `partial=True` in the serializer so that you can update certain fields without requiring the entire object to be sent. Additionally, `get_object_or_404` correctly handles errors and verifies resources on the endpoint, ensuring robustness with possibly missing records.

Remove Race Result Endpoint

The `remove_race_result` method deletes a given race result by ID. It checks whether the object exists (`get_object_or_404`) and then removes it from the database. Endpoint returns the ID of

the deleted result and is transparent to the user. Because it is simple and focuses on deleting records in one go, it is ideal for data removal that requires accuracy.

List Riders Endpoint

The `list_riders` endpoint is the most complicated query implementation in the app. It performs several database functions for every rider: total points by accumulation and unique team the rider was in. What is interesting about this endpoint is that it creates the data structure manually, bringing together data from multiple models. It's longer and more ambiguous than one complex query but gives you better control over the final data structure and illustrates how to handle complicated data relationships when ORM queries are too restrictive

```
@api_view(['GET'])
@permission_classes([AllowAny])
def list_results_sorted(request, year):
    results =
    Result.objects.filter(race__year=year).order_by('race__category',
    'race__id')
    data = {}

    for result in results:
        category = result.race.category
        race_id = result.race.id
        circuit_name = result.race.circuit.circuit_name

        if category not in data:
            data[category] = {}

        if circuit_name not in data[category]:
            data[category][circuit_name] = []

        data[category][circuit_name].append({
            'race_id': race_id,
            'result_id': result.id,
            'rider': result.rider.name,
            'team': result.team.name,
            'position': result.position,
            'points': result.points,
            'time': result.time,
            'speed': result.speed
        })
    )
```

```
    return Response(data)
```

List Results Sorted Endpoint

The `list_results_sorted` method sorts the race results by year, category and circuit. It pulls up results sorted by year, categorises them by category and circuit, and sorts them by preset filters. Endpoint creates a data hierarchy, gathering data of races, riders, teams, and metrics. It is sophisticated in its level of grouping and data formatting, and it leverages Django's ORM for high-level data manipulation.

Styles

The css styling is copied from the project I found online. This styling is under MIT licence. The source reference is added in the `styles.css` file.

Meeting the requirements

This project meets the requirements and marking requirements of having a web application using Django REST and Django REST framework. It indexes records into a relational database (SQLite3), and allows access to the records through well-specified endpoints to retrieve, insert, delete, and update them. The application has seven endpoints, including listing riders, adding riders, adding, editing and removing race results, and retrieving sorted results. These endpoints provide effective data filtration, grouping, and aggregation.

Additionally, I've built the front end for the application with some basic interface, that allows users to view the race or rider's information via browser.

Serialization is used extensively to validate and model data and guarantee uniformity and accuracy. Endpoints such as `api/add_rider/` and `api/list_riders/` are serialized when the data is passed in and out. POST requests permit the generation of new records, satisfying the need to maintain records dynamically.

```
class ResultDetailSerializer(serializers.ModelSerializer):
    race = RaceSerializer()
    rider = RiderSerializer()
    team = TeamSerializer()
    result_id = serializers.IntegerField(source='id')
```

```
class Meta:  
    model = Result  
    fields = ['result_id', 'race', 'rider', 'team', 'position',  
              'points', 'time', 'speed']
```

```
class RiderSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Rider  
        fields = '__all__'
```

It has a Python script to import data from CSV files into the database, so you can load and process large amounts of data. The data model contains things such as Circuit, Rider, Team, Race and Result which actually represent the dataset. Migrations create and maintain the database schema, and unit tests verify endpoint functionality to ensure consistency and accuracy.

The project follows Django standard protocols (URL routing, serializers, validators). It is able to create a scalable, functional REST API with full database support, meeting all stated requirements and markdown criteria.