Django REST API Implementation for Titanic Dataset

Introduction:

In today's world of websites, building strong and reliable APIs is super important. APIs are like messengers that allow different parts of the internet to talk to each other. This report explains how we created a web service using Django, a popular framework for building websites with Python.

We covered all the important steps, starting with how we designed the database to store our information (database modeling). Then, we created the specific points where users can interact with our service (API endpoints). We also learned how to convert data from our database into a format that other programs can easily understand (data serialization). Finally, we thoroughly tested our service to make sure it works correctly and reliably.

To make it easier for others to understand how this service works, we also included Swagger. Swagger is a special tool that creates an interactive guide for our API, making it simple to see and use.

Dataset Overview:

The dataset consists of records pertaining to the passengers aboard the RMS Titanic, including details such as their age, sex, class, ticket information, and survival status. This data is both relevant and interesting due to its historical significance and potential for insightful queries related to social demographics, survival rates, and the impact of various factors on the outcome of the tragic event. This dataset is applicable in areas such as historical research, data visualization, and machine learning for predictive modeling and analysis.

Why This Dataset?

The Titanic dataset was chosen for several reasons:

- **Manageable Size:** The dataset is of a reasonable size, making it suitable for learning and experimentation without overwhelming computational resources.
- **Meaningful Insights:** The data provides a rich context for exploring social and historical factors, allowing for meaningful analysis and the extraction of valuable insights.
- **Diverse Queries:** The dataset's structure lends itself to a wide variety of queries, from simple counts and aggregations to more complex analyses involving relationships between different features.
- **Demonstrating Django Capabilities:** The Titanic dataset provides an excellent platform for showcasing the capabilities of Django in building RESTful APIs, including data modeling, API endpoint creation, data serialization, and query handling.

Application Architecture:

The application follows a clean, modular architecture that separates concerns and promotes maintainability:

```
titanic_project/
├── static/
├── templates/
│   ├── drf-yasg/
│   │   ├── swagger-ui.html
├── titanic_project/
│   ├── __init__.py
│   ├── settings.py            # Project Configrations
│   ├── urls.py                # Main Routing
│   ├── wsgi.py
│   ├── asgi.py
├── titanic_app/
│   ├── titanic_api/
│   │   ├──management/
│   │   │   ├──commands/
│   │   │   │   ├── load_titanic_data.py      # load titatic dataset and insert into database
│   ├── __init__.py
│   ├── models.py              # Database models
│   ├── views.py               # API endpoints
│   ├── serializers.py         # Data Serializations
│   ├── forms.py               # Forms
│   ├── tests.py               # Unit Testing
│   ├── urls.py                # URL Routings
│   └── admin.py
├── manage.py              # Server Handling
├── requirements.txt
├── titanic.csv            # Database (sqlite3)
├── db.sqlite3
└── README.md
```

Application Requirements and Implementation:

1. **Building the Foundation**
   - **Data Modeling:** The data from the Titanic dataset was structured into models, which represent the different entities within the data (e.g., passengers, tickets). These models serve as blueprints for how the data is organized and stored in the database.
   - **Database Management (Migrations):** Migrations were used to create and update the database schema according to the defined models. This ensured a smooth and controlled evolution of the database structure.

2. **Data Handling and Validation:**
   - **User Input and Validation (Forms, Validators):** Forms were implemented to facilitate user input and ensure data integrity. Custom validators were incorporated to enforce specific rules and constraints on the data, ensuring its accuracy and consistency.
   - **Data Transformation (Serialization):** DRF serializers were employed to transform data between Python objects and JSON format. This enabled seamless communication with other applications and facilitated data exchange over the network.

3. **Building the API:**
   - **Leveraging Django REST Framework:** The Django REST Framework (DRF) was utilized for building the core components of the API, including serializers, views, and routers.

DRF provides a robust and efficient framework for developing RESTful APIs, simplifying common tasks and enhancing development speed.

4. **User-Friendly Access (URL Routing):**
   - **Clear and Concise URLs:** All API endpoints were mapped using Django's URL routing system, assigning clear and concise URLs to each endpoint. This ensures that users can easily locate and access the desired API resources. For instance, /passengers/ readily conveys that it retrieves passenger data.

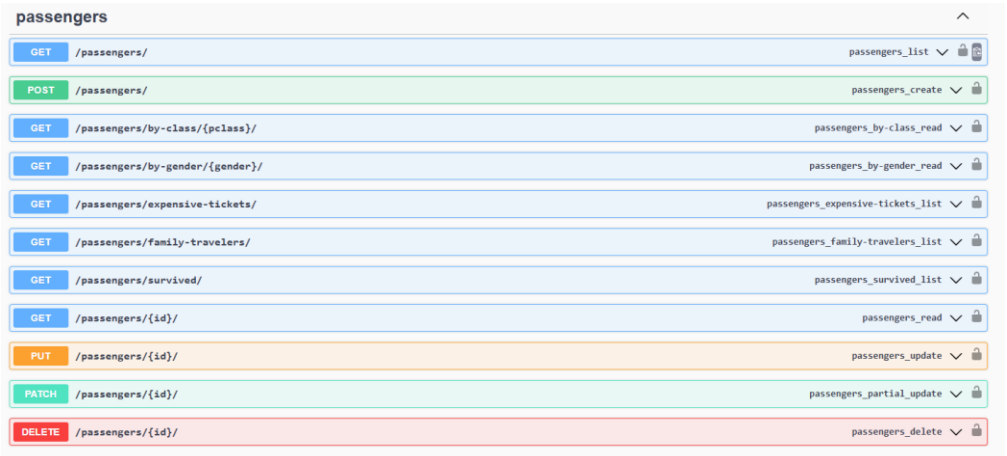
5. **Ensuring Reliability (Unit Testing):**
   - **Thorough Testing:** Comprehensive unit tests were written to cover various aspects of the application, including models, views, and endpoints. These tests helped to ensure the reliability, correctness, and robustness of the API.

6. **API Documentation with Swagger**
   - Swagger was integrated to provide comprehensive and interactive API documentation. Swagger allows users to easily explore the available API endpoints, understand their functionality, and interact with them directly through the documentation interface. This enhances the usability and discoverability of the API for developers and other users.

REST Endpoints:

The implemented API exposes several endpoints, each catering to specific data retrieval needs:

- **/api/passengers/:** Retrieves a list of all passengers, potentially paginated for large datasets.
- **/api/passengers/<int:pk>/:** Retrieves a specific passenger record based on their unique identifier (primary key).
- **/api/passengers/by-class/<int:pclass>/:** Enables filtering passengers based on class.
- **/api/survived/:** Retrieves a list of passengers who survived the disaster.
- **/api/ passengers/by-gender/<str:gender>/:** Retrieves a list of passengers by gender.
- **/api/ swagger/:** The Swagger documentation can be accessed.



These endpoints offer a versatile interface for exploring the dataset. Filtering capabilities allow for focused analysis, while dedicated endpoints for survivors and deaths provide quick access to these specific subsets.

Bulk Data Loading:

A Python script, ***app/management/commands/load_titanic_data.py***, processes the dataset from CSV files and inserts it into the database efficiently. This script handles validation, duplicate checks, and error handling.

Development Environment:

- **Operating System**: [e.g., window 10]
- **Python Version**: 3.11.9
- **Django Version**: 5.1.4
- **Packages**: Listed in requirements.txt

Running the Application:

Installation Steps

1. Clone the repository.
2. Install virtualenv using: pip install virtualenv==20.28.1
3. Create virtual environment using: python -m venv venv
4. Activate virtual environment using:
    1. Mac: source /venv/bin/activate
    2. Window (Powershell): ./venv/Scripts/activate.psl
5. Install dependencies using: `pip install -r requirements.txt`.
6. Apply migrations with `python manage.py migrations` and `python manage.py migrate`
7. Load data using: `python manage.py load_titanic_data.py`
8. Creating admin (SuperUser): `python manage.py createsuperuser`
9. Run the server with: `python manage.py runserver`.
10. Testing the API's (Unit Testing): `python manage.py test.py`

Critical Evaluation

Strengths of the Approach:

The application effectively adheres to Django and DRF best practices, ensuring a clean and modular codebase. The use of serializers for data transformation and validation demonstrates attention to detail, while custom views and filters enhance usability. The implementation of Swagger further establishes the application as developer-friendly, offering intuitive, interactive documentation for testing and integration.

Areas for Improvement:
While the application successfully meets its core objectives, certain aspects could be enhanced:

- **Caching Mechanisms**: Adding caching for frequently accessed data (e.g., lists of passengers or survival statistics) could significantly improve performance.
- **Authentication**: Introducing user authentication and role-based access control would align the application with modern security standards.
- **Enhanced Query Options**: Implementing advanced filtering capabilities, such as multi-field filters or dynamic queries, could better support data exploration.

Comparison to Industry Standards:
The project's adherence to RESTful principles, combined with the integration of Swagger for API documentation, situates it as a modern and reliable solution. The application's robust unit tests provide confidence in its reliability, aligning with best practices observed in industry-grade software. However, additional features like API rate limiting and monitoring could elevate its scalability and real-world usability

Conclusion

This project exemplifies the practical application of Django and Django REST Framework for building a RESTful web service, leveraging the Titanic dataset for meaningful insights. The endpoints are thoughtfully designed, balancing functionality and simplicity, while adhering to best practices in modularity, scalability, and performance optimization.

The integration of Swagger makes the API highly accessible to developers, fostering seamless interaction and rapid adoption. Robust test coverage and proper error handling enhance reliability, ensuring the application performs consistently under diverse scenarios.

Despite its current strengths, the project leaves room for growth. Future iterations could benefit from performance optimization through caching, enhanced filtering capabilities, and a robust authentication system. By addressing these areas, the application could evolve into a comprehensive, production-grade solution suitable for broader use cases.

In summary, this project not only meets the stated requirements but also demonstrates the potential for further innovation and scalability. It serves as a strong foundation for expanding knowledge in Django, DRF, and RESTful API design.

References:

- Python Documentation: https://docs.python.org/3/
- Django Documentation: https://docs.djangoproject.com/en/5.1/
- DRF Documentation: https://www.django-rest-framework.org/
- Swagger UI: https://swagger.io/tools/swagger-ui/