Midterm Report

This 2,314-word report consists of:

- **Technical Content**: ~2,000 words covering implementation and analysis
- **Requirements Mapping**: ~300 words including criteria reference lines under headings and compliance section

1. Introduction

This project implements a REST API using Django and Django REST Framework for a bike store management system. The application manages bike inventory, customer orders, store operations, and staff management across multiple retail locations.

The dataset comprises **approximately 8,000 records** across multiple CSV files covering bikes, brands, categories, customers, stores, staff, orders, and inventory management. Key features include **bulk data loading**, comprehensive CRUD operations, **inventory management** with **stock tracking**, order processing with **automatic stock updates**, and **advanced analytics** for order and staff performance.

2. Dataset Understanding & Preparation

[Addresses Marking Criteria: "Report discusses the dataset - what's interesting about it?"]

The source dataset consists of multiple CSV files containing bike store operational data with approximately 8,000 records total. The primary entities include bikes, brands, categories, customers, stores, staff members, orders, and stock levels across different store locations.

What makes this dataset particularly interesting is its realistic representation of a multi-location bike retail business. The data maintains excellent referential integrity with proper foreign key relationships between entities. Product information includes consistent pricing across different model years, customer data spans various geographic locations, and inventory levels demonstrate realistic distribution patterns across store locations.

The main challenge involved understanding the **business relationships between entities** - for instance, how staff hierarchies work with manager relationships, and how inventory tracking operates across multiple store locations. During analysis, I discovered that orders were distributed across different time periods, providing good test cases for date-based filtering and analytics.

The data quality eliminated the need for extensive cleaning operations, allowing focus on Django development rather than data preprocessing. However, I did implement validation checks during bulk loading to ensure data integrity, particularly for foreign key relationships and stock quantities.

3. Model & Database Design

[Addresses Marking Criteria: "Database/Model design is appropriate for the data provided" & "Application of the techniques taught"]

The database schema implements a normalized design for bike retail operations through eight interconnected models: Brand, Category, Product, Store, Customer, Staff, Order, and OrderItem, plus a Stock model for inventory management.

The Product model serves as the central entity, containing name, model_year, and price fields with foreign key relationships to Brand and Category. I chose DecimalField for price to ensure precise financial calculations.

Customer and Store models implement complete address information with proper field validation. The Customer model uses unique email constraints and includes database indexes on email and name fields for query optimization. These indexes significantly improved search performance when filtering orders by customer.

The Staff model includes a **self-referencing** foreign key for manager relationships, enabling hierarchical staff structures. The active boolean field supports staff status management without data deletion.

Order and OrderItem models implement the classic e-commerce pattern with proper foreign key relationships. The Order model includes order_status choices defined in constants.py. The custom update_items() method provides business logic for inventory management with select_for_update() ensuring thread-safe stock operations - this was crucial for preventing overselling scenarios.

The Stock model creates many-to-many relationships between stores and products with quantity tracking. The unique_together constraint prevents duplicate stock entries while database indexes optimize store-based inventory queries.

4. Bulk Data Loading

[Addresses Marking Criteria: "A method/system is provided to load data in bulk" & "Overall database entries below 10,000"]

The bulk data loading implementation utilizes a custom **Django management command** in scripts/load_data.py that processes CSV files in dependency order to maintain referential integrity. The script loads brands, categories, customers, products, stores, staff, stocks, orders, and order items sequentially.

The loading process leverages Django's **bulk_create** method for performance, significantly reducing database queries compared to individual record creation. During development, I found this approach loaded the entire dataset in under 30 seconds compared to several minutes with individual saves.

The script operates within a **transaction.atomic()** block to ensure data consistency, rolling back all changes if any error occurs. Comprehensive error handling includes validation checks to confirm foreign key references exist before creating dependent records. Duplicate records are prevented using **ignore_conflicts=True**, and existing records are skipped by checking primary keys.

Staff loading required a two-pass approach: first creating staff without manager relationships, then updating manager foreign keys to handle self-referencing dependencies. This solution emerged after initial attempts failed due to circular reference issues.

The script optimizes performance by **pre-fetching foreign key mappings** into dictionaries, minimizing database queries during loading. Progress indicators provide feedback on loading status, which proved helpful during development when troubleshooting data issues.

5. REST API Implementation

[Addresses Marking Criteria: "Application implements all REST endpoints required" & "Report discusses the endpoints - what's interesting about them, how complex is the query?" & "Application implements REST endpoints as links"]

The REST API provides comprehensive endpoints for bike store management operations, leveraging Django REST Framework's capabilities for **complex business logic and analytics**. The endpoints utilize optimized querysets with select_related and prefetch_related to minimize database queries and transaction.atomic() and select_for_update() to update stocks correctly.

1. **POST /api/orders/create** (OrderCreateAPIView): Creates new orders with associated order items. The OrderCreateSerializer validates customer, store, and staff relationships, ensuring sufficient stock through thread-safe operations using select_for_update(). It enforces business rules such as preventing duplicate products and handling insufficient stock scenarios with HTTP 409 Conflict responses.
2. **PATCH /api/orders/update/{order_id}/** (OrderUpdateAPIView): Updates existing orders with partial changes, supporting modifications to order status, dates, and order items. The OrderUpdateSerializer validates updates with thread-safe inventory operations. Invalid updates return appropriate error responses.
3. **GET /api/orders/** (OrdersListAPIView): Retrieves filtered orders with query parameters including customer_id, store_id, staff_id, date ranges,

order_status, and amount filters. The endpoint uses DjangoFilterBackend for dynamic filtering and optimizes queries with select_related to include related data. During testing, I found the delayed_orders filter particularly useful for identifying fulfillment issues.

4. **GET /api/orders/weekday-stats/** (OrdersWeekdayAnalysisAPIView): Provides analytical insights into order distribution by weekday, including total orders, revenue, and average order value. It supports the same filtering parameters as OrdersListAPIView, enabling granular analysis. The endpoint aggregates data using Django ORM's annotate functionality.
5. **GET /api/staff/top-contributors/** (StaffTopContributorsAPIView): Ranks active staff based on performance metrics including order count, total revenue, and a computed performance score. Query parameters include limit (1-50), period filters, and minimum order thresholds. The performance score calculation weighs both volume and value metrics.
6. **GET /api/bikes/brand/{brand_id}/** (BikesByBrandAPIView): Retrieves bikes filtered by brand, with additional parameters for price ranges, categories, model years, and stock availability. The endpoint uses select_related for optimized queries.

All endpoints implement proper HTTP status codes, comprehensive error handling, and consistent JSON response formats. Swagger documentation via drf-yasg enhances API usability, accessible from the main page.

6. Unit Testing

[Addresses Marking Criteria: "Unit testing is included" & "Application makes use of functionality of Django as described in class"]

The testing strategy provides **comprehensive coverage** using Django's testing framework enhanced with model_bakery for realistic test data generation. Test files are organized in the tests file, covering model validations, API endpoints, and business logic scenarios.

Model Testing validates field constraints, relationship integrity, and custom business logic, such as the Order.update_items() method for thread-safe stock management. Tests ensure accurate inventory calculations and maintain referential integrity under various scenarios.

API Endpoint Testing **covers all CRUD** and analytical endpoints, testing successful operations, validation errors, and edge cases. Specific tests include:

- OrdersListAPIView: Verifies filtering by various parameters and error handling for invalid inputs
- OrdersWeekdayAnalysisAPIView: Tests weekday statistics with date range filtering and empty result validation
- StaffTopContributorsAPIView: Validates staff ranking with performance scores and filter parameters
- BikesByBrandAPIView: Confirms correct bike retrieval with multiple filter combinations
- OrderCreateAPIView and OrderUpdateAPIView: Test order operations, stock updates, and error cases

Test Data Generation uses **model_bakery** to create complex test scenarios, including brand-category-product relationships and staff hierarchies. The baker.make method ensures realistic test data, with freezegun controlling date-based logic for delayed order tests.

Test Organization structures tests in focused classes with **setUpTestData** for shared setup and helper methods for reusable logic.

The test suite achieves comprehensive coverage of business logic, error handling, and data integrity requirements.

7. Advanced Techniques & Best Practices

[Addresses Marking Criteria: "Advanced web programming and python techniques, beyond the course content, are used" & "Code is modular and well organised" & "Code is clean (not verbose)" & "Code is functional"]

The implementation demonstrates several advanced Django and DRF techniques beyond basic CRUD operations:

Thread-Safe Operations: The Order.update_items() method uses select_for_update() for inventory management, preventing race conditions in concurrent scenarios. This was essential for reliable e-commerce functionality.

Query Optimization: Strategic use of select_related and prefetch_related in endpoints like OrdersListAPIView minimizes database queries. Database indexes on frequently queried fields and unique_together constraints prevent data duplication.

Serializer Customization: Custom serializers handle complex validation, nested relationships, and computed fields. The performance score calculation in

TopContributingStaffStatsResponseSerializer demonstrates advanced field computation.

API Documentation: Swagger integration with drf-yasg provides detailed endpoint documentation, including query parameters and response schemas, improving developer experience.

Filtering and Analytics: DjangoFilterBackend enables dynamic filtering, while custom analytics endpoints implement aggregation queries for business intelligence.

Error Handling: Custom CustomValidation exception and detailed error responses enhance API reliability and debugging capabilities.

8. Requirements Compliance & Critical Evaluation

8.1 Requirements Mapping

This section explicitly demonstrates how the application meets the coursework requirements:

R1: Basic Django Functionality

- **(a) Models and Migrations**: Eight interconnected models (Brand, Category, Product, Store, Customer, Staff, Order, OrderItem, Stock) with proper field types, constraints, and relationships. Migrations handle database schema creation and updates.
- **(b) Forms, Validators, and Serialization**: Custom serializers (OrderCreateSerializer, OrderUpdateSerializer, etc.) implement validation logic for business rules. Field-level and object-level validation ensure data integrity.
- **(c) Django REST Framework**: All endpoints use DRF's APIView classes, serializers, and response formats. Proper HTTP status codes and error handling implemented throughout.
- **(d) URL Routing**: urls.py files implement clean URL patterns with proper namespacing and parameter handling for RESTful endpoints.
- **(e) Unit Testing**: Comprehensive test suite covering all endpoints, model validation, and business logic using Django's testing framework.

R2: Appropriate Data Model The normalized database schema appropriately models bike retail operations with proper relationships, constraints, and indexes. The model design supports complex queries while maintaining data integrity.

R3: REST Endpoints Implementation Six endpoints demonstrate varying complexity: basic CRUD operations (order creation/updates), filtered listings with multiple parameters, and analytical endpoints with aggregations and computed fields.

R4: Bulk Loading Implementation Custom Django management command processes 8,000+ records efficiently using bulk_create operations, transaction management, and proper dependency ordering.

8.2 Critical Evaluation

The implementation successfully demonstrates comprehensive Django REST Framework usage with a normalized database schema, thread-safe inventory management, and advanced analytics endpoints. The API design balances retail business requirements with RESTful conventions.

Strengths: Analytical endpoints provide valuable business insights leveraging Django ORM's aggregation capabilities. The weekday analysis reveals ordering patterns that could inform staffing decisions, while staff performance metrics enable management reporting.

Comprehensive testing with model_bakery ensures reliable validation of business logic and edge cases. Thread-safe stock management prevents overselling, which is critical for e-commerce reliability.

The bulk loading system efficiently handles large datasets while maintaining data integrity through transaction management and validation.

Areas for Enhancement: The current structure works well for the project scope but could benefit from modularization for larger applications. Separating business logic into service layers would improve maintainability.

Stock management could be enhanced with inventory reservation mechanisms for high-concurrency scenarios, though the current implementation handles typical use cases effectively.

Analytics endpoints could be extended with additional metrics like product popularity trends or customer retention analysis to provide deeper business insights.

The API documentation is comprehensive, but it can benefit from detailed explanation for each api.

9. Setup & Execution Guide

[Addresses Marking Criteria: "Report includes necessary run info such as OS, Python version and login credentials" & "Application runs as required including seeding the db"]

9.1 Development Environment

- **Operating System**: Windows 11
- **Python Version**: 3.10.0
- **Database**: SQLite (included `db.sqlite3` file with pre-loaded data)

9.2 Installation Instructions

1. **Environment Setup**:
    - Download the zip folder and extract the project folder to a known location (don't change the folder name)
    - Open the extracted file named "Advance_Web_Development" in VS Code.
    - Open **PowerShell** and make sure you are in "Advance_Web_Development" directory
    - **(Optional) If you encounter an authorization error, run:**

    ```
    powershell
    # It bypasses authorization for current terminal life span.

    Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
    ```

    - Create virtual environment at the top level:

    ```
    powershell

    python -m venv env
    ```

- Activate the virtual environment:

```
powershell

.\env\Scripts\activate
```

- Navigate into the main project directory:

```
powershell

cd midterm
```

2. **Install Dependencies**:

```
powershell
pip install -r requirements.txt
```

3. **Run the Application**:

```
powershell
python manage.py runserver
```

- Open in browser: http://127.0.0.1:8000

4. **Reimport Data (Optional)**:

```
powershell
del db.sqlite3
python manage.py migrate
python manage.py load_data.py

#To access admin you will have to create superuser
```

```
python manage.py createsuperuser

#To run unit test
python manage.py test
```

9.3 Access Information

- **API Root**: http://127.0.0.1:8000/api/
- **Django Admin**: http://127.0.0.1:8000/admin/
- **Admin Credentials**:
  - Username: `admin`
  - Password: `admin123`
  - Email: `admin@outlook.com`

9.4 Testing

- Run all tests: `python manage.py test`
- Tests are in the `tests/` directory
- Uses `model_bakery` and `freezegun` for test data and date mocking

9.5 Key Dependencies

The project uses Django 5.2.1, Django REST Framework 3.16.0, drf-yasg for API documentation, model-bakery for testing, and other supporting libraries as specified in requirements.txt.

10. Deployment & Publishing

[Addresses Marking Criteria: "Bonus points if you deploy your own app using AWS, Digital ocean, etc."]

The Django 5.2 application has been successfully deployed on **PythonAnywhere** using Python 3.10. The deployment process involved setting up a virtual environment, configuring Django settings for production (ALLOWED_HOSTS, static files), running database migrations, and creating a superuser account. PythonAnywhere was chosen for its Django-optimized environment, built-in HTTPS support, and simplified deployment workflow compared to alternatives like AWS EC2 or DigitalOcean. The application is fully functional with all features including user authentication, admin interface, and static file serving working correctly in the production environment.

🔗 **Live Production URL**: https://asadullahjan.pythonanywhere.com/

🔗 **Admin Interface**: https://asadullahjan.pythonanywhere.com/admin/

10.1 Production Credentials

- Username: `admin`
- Password: `admin123+shop`
- Email: `admin@outlook.com`