DRF React Gems - E-commerce Platform Documentation

Project Overview

DRF React Gems is a full-stack e-commerce platform built with Django REST Framework backend and React frontend. The platform implements user authentication, shopping cart functionality, wishlist management, payment processing, order history tracking, and asynchronous email notifications using Celery and Redis.

Technology Stack

- Backend: Django, Django REST Framework
- Frontend: React
- Database: PostgreSQL
- Authentication: JWT (JSON Web Tokens)
- Cloud Services: Azure (hosting), Redis Cloud (caching)
- Media Storage: Cloudinary
- Frontend Hosting: Firebase
- Error Monitoring: Sentry
- Email Service: Gmail SMTP
- Background Tasks: Celery with Redis

Live URLs

- Main Application: https://drf-react-gems.web.app
- Admin Panel: https://drf-react-gems-f6escmbga4gkbgeu.italynorth-01.azurewebsites.net/admin
- API Documentation: https://drf-react-gems-f6escmbga4gkbgeu.italynorth-01.azurewebsites.net

Architecture and Implementation

Django Framework Configuration

The Django framework is configured with Django REST Framework, JWT authentication, CORS headers, and PostgreSQL database. The complete configuration is located in server/src/settings.py.

Application Structure

- The application contains 16 independent class-based views

- The application contains 12 independent models
- The application contains 7 forms implemented with DRF serializers and React components
- The frontend contains 14 page components

Database Design

Database Technology

The system uses PostgreSQL as the primary database for all application data storage.

Data Normalization

The database follows standard normalization principles:

First Normal Form (1NF): Each table column holds atomic values. Example: The products_color table has a single column for color name with one color per row.

Second Normal Form (2NF): All non-key attributes depend on the entire primary key. Example: In products_inventory table, stock amount and price are linked to specific product-size combinations.

Third Normal Form (3NF): No transitive dependencies exist. Product attributes like color, metal, and stone are stored in separate tables and referenced by foreign keys.

Media Storage

User profile photos are stored using Cloudinary cloud storage service. Implementation is in server/src/accounts/models/user_photo.py.

Frontend Implementation

UI/UX Design

- Custom SCSS modules provide component-based styling for modularity
- Fully responsive user interface adapts to various screen sizes
- Reusable React components ensure consistency
- Accessible navigation with consistent interactive elements including buttons, forms, and popups

Authentication System

Authentication Features

The authentication system uses JWT tokens and includes:

- User registration via UserRegisterView

- User login via UserLoginView
- User logout via UserLogoutView
- Account deletion via UserDeleteView
- Password change functionality via UserPasswordChangeView

Custom Authentication Backend

A custom authentication backend (CustomAuthBackendBackend) allows users to authenticate using either email or username. Implementation is in server/src/accounts/authentication.py.

Password Reset System

Password Reset Process

The system implements a two-step password reset process:

Step 1: User requests password reset via email through UserPasswordResetRequestView
Step 2: User confirms reset using token-based validation through UserPasswordResetConfirmView

Password Reset Features

- Token-based verification using Django's built-in token generator
- URL-safe base64 encoding for user identification
- Automated email delivery via Gmail SMTP with HTML templates
- Password strength validation and confirmation matching
- Implementation in server/src/accounts/serializers/user_credential.py

Access Control System

Public Access

Anonymous users can:

- Browse all products
- Use guest shopping cart stored in browser Local Storage
- Use guest wishlist stored in browser Local Storage

Private Access

Authenticated users can access:

- Checkout process
- Account management

- Order history
- Permanent shopping bag storage
- Permanent wishlist storage

Admin Access

Admin page access is restricted to Order group administrators for sending abandoned cart reminder emails.

Review Moderation

- Regular users see only approved product reviews
- Order group users with products.approve_review permission can view all reviews including unapproved ones

Admin Interface

Custom Admin Features

The admin interface uses the Unfold framework with:

- Custom themes and styling (server/src/unfold.py)
- Organized sidebar navigation with collapsible sections
- Permission-based access control
- Different admin sections based on user permissions

Admin Customizations

The admin interface includes 5 custom options:

1. List filters by stone and color
2. List display with image previews
3. Record ordering functionality
4. Search fields for product attributes
5. Inline editing of related inventory

Implementation is in server/src/products/admin.py.

Admin Groups and Permissions

Admin Group Structure

Three admin groups with different permission levels:

1. Superuser Group:

- Complete system administration
- Full CRUD operations on all models
- Access to all admin sections including Users and Groups
- Login: super_user@mail.com
- Password: !1Aabb

2. Inventory Group:
   - Full CRUD access to products, inventory, and attributes
   - Access to Products and Product Attributes sections
   - Login: inventory_user@mail.com
   - Password: !1Aabb
3. Order Group:
   - Can approve and disapprove customer reviews
   - Access to Product Reviews section
   - Can send abandoned cart reminder emails
   - Login: order_user@mail.com
   - Password: !1Aabb

Automated Setup

Admin groups and users are created automatically via management command in server/src/accounts/management/commands/create_roles.py.

Exception Handling and Validation

Server-Side Validation

- Try-except blocks with DRF exception classes for authentication and critical operations
- Password validation in server/src/accounts/validators/password.py
- Payment and order validation in server/src/orders/services.py
- Model validation in server/src/accounts/validators/models.py

Client-Side Validation

- Real-time form validation for all forms (registration, login, delivery, payment, password update)
- Visual feedback with color-coded input fields (green for valid, red for invalid, blue for focus)
- Server-side error integration maps backend errors to correct form fields
- Custom React hooks and validation helpers prevent invalid requests

Asynchronous Processing

Asynchronous Views

1. notify_users_they_have_uncompleted_orders:
    - Allows Order group admins to send reminder emails
    - Targets shopping bags older than one day
    - Demo available at https://drf-react-gems.web.app/admin-page
    - Requires Order User authentication
2. send_email:
    - Handles email notifications via Gmail SMTP
    - Uses Google App Password authentication
    - Implementation in server/src/common/views.py

Background Tasks with Celery and Redis

Background tasks include:

- User greeting emails sent on registration (server/src/accounts/signals.py)
- Scheduled task to mark old orders as completed (server/src/orders/tasks.py)
- Review approval notification emails (server/src/products/signals.py)

Testing

Test Coverage

The application includes 66 unit and integration tests with 74% code coverage. Coverage report is available at https://beatrisilieva.github.io/drf-react-gems/coverage-report/index.html.

To run tests: coverage run manage.py test && coverage report

Deployment Architecture

Deployment Services

- Backend hosting: Azure
- Cache service: Redis Cloud
- Frontend hosting: Firebase
- Coverage reports: GitHub Pages
- Media storage: Cloudinary

User Model Extension

Custom User Implementation

- Email used as primary login identifier
- Unique username requirement
- Additional fields for marketing consent
- User profile model stores personal and shipping information
- User photo model handles profile pictures with cloud storage
- Automatic profile and photo creation using Django signals
- One-to-one relationships established via signals in server/src/accounts/signals.py

Object-Oriented Design Principles

Design Patterns and Principles

1. Inheritance:
    - All product categories inherit from abstract base product model
    - Shared fields defined in server/src/products/models/base.py
2. Open/Closed Principle:
    - New categories extend existing logic without modification
    - Implementation in server/src/products/models/product.py
3. Polymorphic Relations:
    - Single inventory and review system for all product types
    - Uses Django's GenericForeignKey
    - Each product has three sizes with individual prices and quantities
    - Implementation in server/src/products/models/inventory.py and review.py
4. Abstraction:
    - Product fetching logic abstracted into BaseProductManager
    - Handles prefetch_related, values, and annotations
    - Provides price ranges and average ratings
5. Data Encapsulation:
    - Business logic in dedicated service classes
    - Validation in model managers
6. Cohesion and Loose Coupling:
    - Each Django app encapsulates distinct business domain
    - Apps: accounts, products, orders, shopping_bags, wishlists
    - Apps are primarily independent from each other

7. Code Quality:
    - Python code adheres to PEP 8 standards
    - JavaScript uses ESLint and Prettier for formatting

Installation Requirements

Prerequisites

- Python (version 3.8 or higher recommended)
- Node.js (version 14 or higher recommended)
- PostgreSQL database
- Redis server
- Git version control

Backend Environment Variables

Required environment variables for server configuration:

- SECRET_KEY: Django secret key (required)
- ALLOWED_HOSTS: Comma-separated list of allowed hosts
- CORS_ALLOWED_ORIGINS: Frontend URL for CORS
- CSRF_TRUSTED_ORIGINS: Frontend URL for CSRF
- FRONTEND_URL: Frontend URL for password reset emails
- DB_NAME: PostgreSQL database name (required)
- DB_USER: PostgreSQL username (required)
- DB_PASS: PostgreSQL password (required)
- DB_HOST: Database host (127.0.0.1 for local)
- DB_PORT: Database port (5432 default)
- CLOUD_NAME: Cloudinary cloud name (required)
- CLOUD_API_KEY: Cloudinary API key (required)
- CLOUD_API_SECRET: Cloudinary API secret (required)
- CELERY_BROKER_URL: Redis URL for Celery broker
- CELERY_RESULT_BACKEND: Redis URL for Celery results
- EMAIL_HOST: Email server host
- EMAIL_PORT: Email server port
- EMAIL_HOST_USER: Email account
- DEFAULT_FROM_EMAIL: Default sender email

- SERVER_EMAIL: Server email address
- EMAIL_HOST_PASSWORD: Email SMTP app password (required)
- SENTRY_DSN: Sentry error monitoring DSN
- DEBUG: Debug mode (True for development)

Frontend Environment Variables

Required environment variables for client configuration:

- VITE_APP_SERVER_URL: Backend server URL (http://localhost:8000 for local development)

Installation Steps

1. Clone repository: git clone https://github.com/BeatrisIlieva/drf-react-gems.git
2. Navigate to project: cd drf-react-gems
3. Create backend .env file in server directory
4. Setup Python virtual environment: python3 -m venv .venv
5. Activate virtual environment: source .venv/bin/activate
6. Install Python dependencies: pip install -r requirements.txt
7. Run database migrations: python manage.py migrate
8. Populate database: python manage.py setup_database
9. Start backend with honcho: export PORT=8000 && honcho start
10. Create frontend .env.development file in client directory
11. Install frontend dependencies: npm install
12. Start frontend development server: npm run dev

Database Population

The setup_database management command performs:

- Creates all products with categories, collections, colors, metals, stones, and inventories
- Adds customer reviews with ratings and comments
- Creates admin users with different roles and permissions
- Creates a superuser with full system access

Project Repository

GitHub Repository: https://github.com/BeatrisIlieva/drf-react-gems

License