# Approach to Solving the Inventory Management System Problem

The Inventory Management System provides an API for managing products and categories with user authentication and role-based access. The backend is built with Django and Django REST Framework (DRF), utilizing JWT for authentication, while the frontend is implemented with React, providing an interactive UI for admins and regular users.

## Backend Design

### Models

1. **CustomUser**:
    - Built by extending Django's `AbstractUser`, this model includes an additional field, `is_admin`, to allow for role-based access within the system.
    - Using `is_admin` gives us flexibility in granting admin access independently of Django's superuser, allowing the frontend to handle authorization without additional backend access.
2. **Category**:
    - Each `Category` contains a unique `name` and an optional `description`. This model organizes products and is referenced by the `Product` model as a foreign key.
3. **Product**:
    - Each `Product` includes fields for `name`, `category`, `price`, `quantity`, `description`, and `stock_level`.
    - To allow safe deletion of categories without cascading deletions on products, the `category` field uses `on_delete=models.SET_NULL`, setting it to `NULL` when a category is deleted.

### Serializers

1. **CustomUserSerializer**:
    - Handles user creation, including password hashing. `is_admin` is made optional, enabling users to register as admins if specified, creating a flexible user creation process.
2. **CategorySerializer**:
    - Serializes all fields in the `Category` model for CRUD operations.
3. **ProductSerializer**:
    - Extends standard fields with a read-only `category_name` field (through `source='category.name'`) to provide the category's name in the serialized product data, enhancing usability in the frontend.

### Views

1. **Authentication and User Views**:

    - `CustomUserRegistrationView`: Allows users to register, with the option to set the `is_admin` field if needed.
    - `UserInfoView`: Returns current user information, accessible to authenticated users.
2. **Product Management**:
    - **ProductListView**: Displays a list of all products, accessible to authenticated users.
    - **ProductAddView**: Adds a new product or updates an existing product's stock level if the name matches. This helps avoid duplicate entries.
    - **ProductUpdateView**: Updates product details, adjusting the `stock_level` by `quantityChange`. It validates that `stock_level` does not go below 1.
    - **ProductDeleteView**: Deletes a product by ID, only accessible to admins.
    - **ProductsByCategoryView**: Lists products under a specified category, enhancing filtering capabilities.
3. **Category Management**:
    - **CategoryListCreateView**: Lists and creates categories.
    - **CategoryDetailView**: Deletes a category and updates products to set `category=None`. This preserves products while removing the category association.

**URL Configuration**

The backend routes are organized for clarity:

- Authentication endpoints (registration, login, and user info).
- Product and category CRUD endpoints, including specific routes for listing products by category.

## Why Use `is_admin` Instead of Django Superuser

- Using `is_admin` provides flexibility to grant admin rights without needing full superuser permissions, allowing for finer control.
- The system can handle admin privileges directly within the business logic without exposing high-level Django superuser access, which enhances security for basic admin actions.

## Frontend Design

The frontend, built with React, provides the following views:

1. **HomePage**:
    - Displays a welcome message tailored to the user's role (admin or user) and provides navigation links to products and categories.
2. **Category Management**:

    - Admins can view, add, edit, or delete categories. Deletion removes the category association from any linked products, displaying “No Category” instead.
3. **Product Management**:
    - Users can view products, and admins can add, edit, and delete products. For updates, admins specify stock adjustments, where positive values increase and negative values decrease stock, maintaining flexibility in stock management.

### Error Handling

- The backend validates conditions like non-negative stock levels, and meaningful error messages are sent in the response.
- The frontend displays these messages as alerts, ensuring users understand constraints like minimum stock levels and required fields.

### Notification System for Low Stock

Admins receive an alert on the home page if any product’s stock level is critically low, enhancing inventory oversight.

### Additional Details

- **Testing**:
    - Comprehensive tests cover user registration, authentication, and CRUD operations on products and categories. Tests ensure that only admins can perform certain actions and that appropriate errors are returned when validation fails.
- **Requirements**:
    - Dependencies are listed in `requirements.txt`, making it easy to set up the backend environment consistently across systems.

# Settings and Configuration (settings.py)

The `settings.py` file in Django is configured as follows:

- **Installed Apps**:
    - Core apps (`django.contrib.*`) and third-party apps (`rest_framework`, `corsheaders`, `django_extensions`) are included.
    - The custom app, `inventory`, is added to manage the system’s models, views, and serializers.
- **Authentication**:
    - **JWT Authentication**: Implemented using `rest_framework_simplejwt`, supporting secure token-based access for authenticated users.
- **Custom User Model**:

    - `AUTH_USER_MODEL = 'inventory.CustomUser'` replaces the default Django user model with our `CustomUser` model to accommodate `is_admin`.
- **Database**:
    - Uses SQLite for simplicity (`db.sqlite3`), though the configuration can be updated to any other database.
- **Cross-Origin Resource Sharing (CORS)**:
    - Configured with `CORS_ALLOWED_ORIGINS` to accept requests from `localhost:3000`, where the React frontend runs.
- **CORS Middleware**:
    - The `corsheaders` middleware is included to handle cross-origin requests from the frontend.

### Environment Variables (Recommended)

For production, environment variables should be used to handle sensitive data such as `SECRET_KEY` and database credentials. This setup would enhance security and flexibility in deployment.

---

# Testing

Comprehensive tests cover the following areas:

- **User Registration and Authentication**: Tests validate successful registration, login, and access control based on user roles.
- **Product and Category CRUD**: Each CRUD endpoint is tested for both successful operations and validations (e.g., only admins can delete products and categories).
- **Error Handling**: Tests ensure appropriate error responses for invalid data, providing feedback for missing or incorrect fields.