Title: Building a Backend API for a Simple Inventory Management System using Django Rest Framework

Objective:
Develop a backend API for an Inventory Management System that supports CRUD operations on inventory items, integrated with JWT-based authentication for secure access. The system should use Django Rest Framework (DRF) for the API framework, PostgreSQL for the database, Redis for caching, and include unit tests to ensure functionality. Implement proper error handling with appropriate error codes and integrate a logger for debugging and monitoring.

Background:
An inventory management system allows a business to manage its stock of products efficiently. The system must provide endpoints to create, read, update, and delete (CRUD) items in the inventory. To improve performance, frequently accessed items should be cached using Redis. Additionally, JWT authentication will be used to secure access to the API.

Requirements:

- **API Framework:** Use Django Rest Framework (DRF) to create the API.
- **Database:** Use PostgreSQL to store inventory items.
- **Caching:** Use Redis to cache frequently accessed items.
- **Authentication:** Use JWT for securing the API endpoints.
- **Logging:** Integrate a logging system for debugging and monitoring.
- **Testing:** Implement unit tests using Django's test framework to verify the functionality of the API.

Functional Specifications:

1. **Authentication:**
   - **JWT Authentication:** Secure all endpoints using JWT. Users must authenticate to access CRUD operations.
   - **Endpoints:** Include endpoints for user registration, login, and token retrieval (using JWT).
2. **Create Item Endpoint:**
   - **Method:** POST
   - **Path:** `/items/`
   - **Request Body:** JSON object containing item details (e.g., name, description).
   - **Response:** JSON object with the created item details.
   - **Error Codes:**
     - `400:` Item already exists.
3. **Read Item Endpoint:**
   - **Method:** GET
   - **Path:** `/items/{item_id}/`
   - **Response:** JSON object with the item details.
   - **Error Codes:**
     - `404:` Item not found.
4. **Update Item Endpoint:**

    - **Method:** PUT
    - **Path:** `/items/{item_id}/`
    - **Request Body:** JSON object containing updated item details.
    - **Response:** JSON object with the updated item details.
    - **Error Codes:**
        - `404:` Item not found.
5. **Delete Item Endpoint:**
    - **Method:** DELETE
    - **Path:** `/items/{item_id}/`
    - **Response:** Success message.
    - **Error Codes:**
        - `404:` Item not found.
6. **Redis Caching:**
    - Implement caching for the Read Item endpoint to improve performance.
    - Store the item in Redis when it is accessed for the first time.
    - Fetch the item from Redis for subsequent requests.
7. **Database Interaction:**
    - Use Django ORM for interacting with the database.
    - Ensure the database schema is properly defined, and migrations are managed using Django's migration system.
8. **Logging:**
    - Integrate a logging system to track API usage, errors, and other significant events.
    - Use appropriate logging levels (e.g., INFO, DEBUG, ERROR).
9. **Unit Tests:**
    - Implement unit tests for all endpoints using Django's test framework.
    - Ensure tests cover success and error cases.

Constraints:

- Use Django Rest Framework for the API framework.
- Use PostgreSQL or MySQL for the database.
- Use Redis for caching.
- Implement JWT authentication for securing endpoints.
- Write Django ORM queries for database interactions.
- Implement unit tests using Django's test framework.
- Integrate a logging system for monitoring and debugging.

Deliverables:

1. **Modular Django Application:**
    - Implement the application with a modular approach, separating concerns into distinct modules (e.g., models, views, serializers, caching, authentication).
    - Ensure each module has a clear responsibility and is easily maintainable and testable.
2. **Database Schema and Migration Scripts:**
    - Provide Django models and migration scripts for database setup and management.
3. **Redis Caching Implementation:**

    - Integrate Redis caching for the read endpoint, ensuring it is modular and easily configurable.
4. **JWT Authentication Integration:**
    - Implement JWT-based authentication for all endpoints, securing access to the API.
5. **Unit Tests for All Endpoints:**
    - Develop comprehensive unit tests using Django's test framework to cover all API functionalities, ensuring correctness and reliability.
6. **Logging Integration:**
    - Integrate a logger to capture API usage, errors, and other significant events. Use appropriate logging levels.
7. **README File:**
    - Include setup instructions, API documentation, and usage examples to guide users through the installation and operation of the application.
8. **Code Submission:**
    - Submit the complete project as a zip file or provide a link to a version control repository (e.g., GitHub) containing all source code and documentation.