DATABASE SCHEMA FOR INVENTORY MANAGEMENT SYSTEM

Explanation of the Database Schema

1. **User Model**:
   - For authentication and authorization
   - Tracks basic user information and admin status
2. **Product Model**:
   - Stores product information (name, description, price, etc.)
   - Each product has a unique SKU
   - Includes metadata like weight, dimensions, and category
3. **Inventory Model**:
   - Tracks stock levels for each product
   - Includes quantity in stock and quantity reserved for processing orders
   - Has reorder levels and quantities for inventory management
   - Provides properties for available quantity and reorder needs
4. **Order Model**:
   - Tracks customer orders with a unique order number
   - Has various order statuses (pending, processing, shipped, delivered, cancelled)
   - Includes shipping and tracking information
5. **OrderItem Model**:
   - Represents individual items within an order
   - Captures the quantity and price at the time of order

Features of this Schema

- **Simple**: Clear separation of concerns with distinct models for each entity
- **Flexible**: Models include various fields that can be extended
- **Robust**: Includes proper relationships and constraints
- **Ready for Scaling**: Designed with inventory reservation to prevent overselling
- **Supports the Requirements**: Handles order processing, inventory management, and no backorders

AUTHENTICATION AND AUTHORISATION

The flow works like this:

1. Users register an account
2. Users login with email/password and receive tokens
3. The access token is used for regular API calls
4. When the access token expires, the refresh token can be used to get a new pair of tokens
5. Protected endpoints can use the `get_current_user` or `get_current_active_admin` dependencies to verify access

Benefits of JWT Authentication for the Inventory System

Key Advantages

- **Stateless**: No server-side session storage required, reducing database load
- **Scalability**: Easily scales across multiple servers without shared session stores
- **Performance**: Faster authentication checks than database lookups on every request
- **Security**: Tamper-proof encoded user information with digital signatures
- **Mobile-Friendly**: Works seamlessly with mobile apps and API clients
- **Expiration Control**: Built-in token expiration with refresh token pattern
- **Reduced Database Load**: Minimizes authentication queries to the database

Inventory Management API

Key Features:

- Get inventory list with optional filtering (low stock, by product)
- Get inventory details for specific products
- Create inventory records for products
- Update inventory details (reorder levels, quantities)
- Adjust stock levels with reason tracking
- Special endpoint for low-stock reporting

Highlights:

- Proper separation of reserved vs. available stock
- Automatic tracking of restock dates
- Admin-only endpoints for sensitive operations
- Proper error handling for all operations
- Clean schema design with validation

Order Processing API

Key Features:

- Create orders with multiple items
- Background task processing for orders
- List orders for current user
- Admin view of all orders with user details
- Get order details
- Update order status with proper inventory management
- Cancel orders with inventory release

Highlights:

- Complete inventory validation before order creation
- Proper handling of all order states (pending → processing → shipped → delivered)
- Order cancellation at any point before shipping
- Automatic inventory reservation and deduction
- Secure access control (users see only their orders, admins see all)

Key Design Principles

- **Clean Separation of Concerns**: Models, schemas, and routers are well-organized
- **Robust Error Handling**: Proper HTTP status codes and error messages
- **Security First**: JWT authentication and proper authorization checks
- **Data Integrity**: Transaction management to prevent inconsistent states
- **Performance**: Background processing for order handling
- **Flexibility**: Schema design allows for extension without breaking changes

This implementation fully addresses the requirements for a RESTful API with endpoints for placing orders, checking inventory, and updating stock levels, with proper error handling and status codes.

Redis Queue Background Job System

Redis Queue-based background job system for order processing. Here's what I've added:

1. Core Components

- **Redis Queue Configuration** (app/core/queue.py)
  - Connects to Redis server
  - Sets up multiple priority queues (high, default, low)
  - Provides a function to enqueue tasks
- **Worker Module** (app/worker/)
  - Contains the background job functions
  - Handles order processing, inventory management, and error handling
- **Worker Script** (worker.py)
  - Runs the RQ worker process to consume jobs from the queues
  - Can be run in multiple instances for scalability

2. Order Processing Flow

1. When an order is created, it's added to the high-priority queue
2. A worker picks up the order and processes it asynchronously:
   - Updates the order status to PROCESSING
   - Reserves inventory for the items
   - Simulates payment processing
   - Handles success or failure appropriately
3. If successful, the order can move to shipping (which could be another queued job)
4. If failed, inventory is released and the order is marked as cancelled

3. Key Features

- **Separate Queues by Priority**: Different queues for different job priorities
- **Error Handling**: Robust error handling with automatic rollback and recovery
- **Retry Logic**: Retry mechanism for database operations to handle race conditions
- **Logging**: Detailed logging for monitoring and debugging
- **Resource Management**: Proper database session management in worker functions
- **Scalability**: Run multiple workers to scale processing capacity

4. Running the System

1. Start Redis server: `redis-server`
2. Start one or more worker processes: `python worker.py`
3. Start a FastAPI application: `uvicorn app.main:app --reload`

HOW IT ALL COMES TOGETHER:

How These Files Work Together

1. **Router Files** (*`app/routers/order.py`*, *`app/routers/inventory.py`*)
    - Handle HTTP API requests/responses
    - Define the endpoints users/clients can call
    - Perform immediate validation and basic operations
    - **Enqueue** background jobs when needed
2. **Worker File** (*`app/worker/order_tasks.py`*)
    - Contains functions that run **asynchronously** in the background
    - Performs time-consuming operations (payment processing, complex inventory updates)
    - Runs in a separate process from your web server
    - Gets tasks from Redis queue, not from HTTP requests

The Flow in Action

1. User makes API request to create order → *`order.py`* router handles this
2. Router validates data, creates order record, then calls:

```
job = enqueue_task(process_order, new_order.id,
queue_name='high')
```

3. Router returns success response immediately to user
4. **Meanwhile**, a separate worker process picks up the job
5. Worker executes `process_order` function from `order_tasks.py`

This separation improves the system by:

- Keeping API responses fast (not blocked by processing)
- Allowing complex operations to happen asynchronously
- Providing natural retry capability for failed operations
- Enabling better scaling (add more workers when needed)

Redis Time-Based Caching with Active Invalidation

For the Inventory Management System, I've implemented a Redis-based caching strategy that balances performance with data consistency. This approach improves API response times by reducing database load for frequently accessed inventory data.

Caching Strategy

I chose a **TTL (Time-To-Live) caching with active invalidation** approach because:

1. **Redis** provides fast in-memory storage with built-in expiration support
2. **Time-based expiration** ensures cache automatically refreshes (preventing indefinitely stale data)
3. **Different TTLs for different data types** optimize for access patterns:
    - Regular inventory data: 10 minutes TTL
    - Low-stock items: 3 minutes TTL (more frequent updates needed)

Cache Consistency Mechanism

The system maintains consistency between cache and database through:

1. **Write-Through Caching**: Database is always the source of truth
2. **Active Cache Invalidation**: Cache entries are explicitly invalidated whenever inventory data changes:
    - When inventory is directly updated or adjusted
    - When order status changes affect inventory levels (cancellation, processing, shipping)
    - When orders are created or cancelled
3. **Automatic Fallback**: On cache miss, the system retrieves fresh data from database

This mechanism ensures inventory data remains consistent while dramatically improving read performance for inventory checks during order creation and browsing - the most frequent operations in an e-commerce system

---

Scalability & Performance Analysis for Inventory System

Database Connection Pooling Limitations

Problem: My FastAPI application creates new database connections for each request. As user traffic increases, this can exhaust available database connections, causing timeouts and failures.

Proposed Solution: Implement Connection Pooling

- Configure SQLAlchemy to use a connection pool with appropriate settings
- Set maximum connections based on database server capacity
- Implement connection timeout and retry logic
- This ensures efficient database connection reuse instead of creating new connections for every request

Performance Bottlenecks

1. N+1 Query Problem in Order Retrieval

Problem: In the `get_all_orders` endpoint, I'm making additional database queries for each order's items, creating excessive database load.

Solution:

- Implement a GraphQL API for flexible data fetching

2. Missing Database Indexes

Problem: The current schema lacks specific indexes for common query patterns. So my database queries will become increasingly slow as data grows, particularly for order filtering and low-stock calculations.

Solution:

Add composite indexes on frequently filtered columns:

```
CREATE INDEX idx_inventory_low_stock ON inventory(product_id, (quantity_in_stock - quantity_reserved), reorder_level);
CREATE INDEX idx_orders_user_status ON orders(user_id, status, created_at DESC);
CREATE INDEX idx_order_items_product ON order_items(product_id, order_id);
```

- Implement database query monitoring to identify slow queries
- Use partial indexes for specialized query patterns (e.g., only low stock items)

3. Redis Single-Point-of-Failure Architecture

Problem: The system relies on a single Redis instance (though separated into different databases) for both job queues and data caching. This creates a key scalability limitation and reliability risk.

Specific Issues:

- Both queue and cache operations compete for the same Redis server resources
- If the Redis server fails, all background processing and caching fail simultaneously
- Limited capacity for handling high traffic periods
- No redundancy or failover capability

Solution: Implement a distributed Redis architecture with:

- Physically separate Redis servers for queues and caching
- Redundant servers with automatic failover for each function
- Implementation of Redis Sentinel or Redis Cluster for high availability
- Potentially different messaging technology for mission-critical queues

4. Background Worker Limitations

Problem: The current worker implementation lacks critical resilience and observability features necessary for a production system.

Specific Issues:

- **No Retry Policy:** Failed jobs are not automatically retried, resulting in permanent failures even for temporary issues
- **Insufficient Monitoring:** No notification system for worker failures or job errors
- **Limited Visibility:** No metrics or dashboards to track worker health and performance
- **No Dead Letter Queue:** Failed jobs disappear with no record or recovery path

Solution: Enhance the worker implementation with:

- Automated retry policies with exponential backoff
- Integration with monitoring and alerting systems
- Implementation of dead letter queues for failed jobs
- Health check endpoints and performance metrics
- Proper error handling and logging

Explanation of the Authentication Tests

Registration Tests

1. **test_register_user_success**: Makes sure users can sign up successfully when they provide valid information. It checks that the system creates a new account and returns the right data.
2. **test_register_existing_email**: Ensures you can't create two accounts with the same email address. The system should reject the second attempt with an appropriate error message.
3. **test_register_invalid_email**: Verifies that the system catches invalid email formats (like "not-an-email") and rejects them rather than creating accounts with bad data.
4. **test_register_short_password**: Checks that the system enforces password security by rejecting passwords that are too short (less than 8 characters).

Login Tests

5. **test_login_success**: Confirms that a valid user can log in with the correct credentials and receive access and refresh tokens.
6. **test_login_wrong_email**: Makes sure the system properly handles login attempts with email addresses that don't exist in the database.
7. **test_login_wrong_password**: Verifies that the system rejects login attempts when the password is incorrect for an existing account.
8. **test_login_inactive_user**: Tests that inactive accounts can't log in, even with correct credentials. This is important for security when accounts are suspended.

Token Refresh Tests

9. **test_refresh_token_success**: Checks that users can get new tokens using their refresh token without having to log in again. This is crucial for maintaining sessions.
10. **test_refresh_invalid_token**: Confirms that the system properly rejects attempts to refresh tokens using invalid or tampered tokens.

Database Tests

11. **test_password_verification_in_db**: Makes sure passwords are being properly hashed and stored securely, and that the verification process works correctly.
12. **test_new_user_in_database**: Verifies that when a user registers, their information is actually saved in the database with the correct values.

Explanation of the Order Management Tests

Order Creation Tests

1. **test_create_order_success**: Verifies that users can successfully create orders with available inventory. It checks that the API correctly validates the order data, calculates the total price, creates the database records, and queues a background task for processing.
2. **test_create_order_insufficient_stock**: Tests the inventory validation logic by attempting to create an order for more items than are available in stock. It ensures the system properly rejects such orders and provides a clear error message.
3. **test_create_order_nonexistent_product**: Confirms that the system correctly validates product existence, rejecting orders containing products that don't exist in the database with appropriate error messages.

Order Retrieval Tests

1. **test_get_user_orders**: Checks that users can retrieve a list of their own orders, with proper filtering and complete order details including items.
2. **test_get_order_detail**: Verifies that users can access detailed information about a specific order by its ID, including all order items and status information.
3. **test_get_order_unauthorized**: Tests authorization controls by confirming that users cannot access orders that belong to other users, ensuring proper data privacy.

Admin Functionality Tests

1. **test_admin_get_all_orders**: Confirms that admin users can retrieve all orders in the system with complete user and product details, supporting administrative oversight.
2. **test_non_admin_cannot_access_all_orders**: Verifies that regular users cannot access admin-only endpoints, ensuring proper role-based access control.
3. **test_admin_update_order_status**: Tests that admin users can update order status, and that these updates correctly trigger inventory adjustments (like reserving inventory when moving to processing).

Order Cancellation Tests

1. **test_order_cancellation**: Checks that users can cancel their own pending orders, and that cancellation properly releases any reserved inventory back into available stock.
2. **test_cannot_cancel_shipped_order**: Verifies that orders cannot be cancelled after they've been shipped, enforcing business rules about order lifecycle.

Background Processing Tests

1. **test_process_order_background**: Tests the asynchronous order processing functionality, ensuring that background tasks correctly update order status and reserve inventory when payment is successful.
2. **test_process_order_payment_failure**: Confirms that when payment processing fails, the system correctly marks orders as cancelled and releases any reserved inventory.
3. **test_reserve_inventory_race_condition**: Verifies that the inventory reservation system handles concurrent updates correctly, preventing overselling by ensuring availability checks are accurate even under heavy load.

Explanation of the Inventory Management Tests

Inventory Retrieval Tests

1. **test_get_inventory**: Verifies that authenticated users can retrieve a list of all inventory items with product details. It checks that the system returns the correct data structure with both inventory and product information.
2. **test_get_inventory_with_filters**: Tests the filtering capabilities of the inventory endpoint. It verifies that users can filter inventory by product ID and low stock status, ensuring the system correctly identifies and returns only matching items.
3. **test_get_inventory_item**: Confirms that users can retrieve detailed information about a specific inventory item by its ID, with complete product details included.
4. **test_get_nonexistent_inventory_item**: Ensures the system handles requests for non-existent inventory items gracefully, returning an appropriate 404 error rather than crashing.

Inventory Creation Tests

1. **test_create_inventory**: Verifies that admin users can create new inventory records for products that don't already have inventory. It checks that the system properly records stock levels and automatically sets the restock date.
2. **test_create_inventory_nonexistent_product**: Tests that the system prevents creating inventory for products that don't exist, returning a 404 error with an informative message.
3. **test_create_inventory_duplicate_product**: Confirms that the system prevents duplicate inventory records for the same product, maintaining data integrity by returning a 400 error.
4. **test_create_inventory_non_admin**: Ensures that non-admin users cannot create inventory records, verifying that permission controls are functioning correctly.

Inventory Update Tests

1. **test_update_inventory**: Checks that admin users can update inventory details such as stock levels and reorder thresholds, and that these changes are correctly saved in the database.
2. **test_adjust_stock**: Verifies the stock adjustment functionality for increasing inventory quantities, ensuring that the system correctly updates stock levels and restock dates.
3. **test_adjust_stock_negative**: Tests the validation requirements for stock quantity adjustments, confirming that the system enforces proper constraints on input values.
4. **test_adjust_stock_below_zero**: Ensures that the system's validation prevents operations that would result in negative inventory, maintaining data integrity.

Product-Specific Inventory Tests

1. **test_get_product_inventory**: Confirms that users can retrieve inventory information for a specific product by its ID, getting complete inventory details.
2. **test_get_low_stock_items**: Verifies that admin users can retrieve a list of all products with stock levels at or below their reorder thresholds, helping identify items that need replenishment.