

Inventory Management System for B2B SaaS – Case Study

Candidate Name: Samarth Chaugule

Role: Backend Engineering Intern

Title :

This case study demonstrates my approach to backend problem-solving, database design, and API implementation for a B2B inventory management system. The focus is on correctness, scalability, and real-world production considerations.

Part 1: Code Review & Debugging – Solution

Problem Understanding

The given API endpoint is responsible for:

1. Creating a new product
2. Initializing inventory for that product in a warehouse

Although the code compiles successfully, it lacks several technical and business-level safeguards required for a production-ready backend system.

Note:

Even though the sample code is written in Flask, the identified issues and solutions are **framework-independent** and apply to any backend system.

1. Issues Identified

Technical Issues

1. No validation of request data before database operations.
2. No error handling for database failures.

3. Multiple database commits instead of a single atomic transaction.
4. No rollback mechanism in case of partial failure.
5. Price is not handled in a precision-safe manner.
6. Concurrency issues can occur during simultaneous requests.

Business Logic Issues

7. SKU uniqueness is not enforced.
8. Product creation is tightly coupled to a single warehouse.
9. Inventory is created without verifying warehouse validity.
10. Optional or missing fields are not handled gracefully.

2. Impact in Production

1. Invalid or missing input may cause runtime errors.
2. Duplicate SKUs can break product identification across the platform.
3. Partial data creation may occur (product exists without inventory).
4. Multi-warehouse product storage becomes impossible.
5. Price precision errors can affect billing and reporting.
6. High-traffic scenarios can create inconsistent database states.
7. Debugging production issues becomes difficult due to poor error handling.

3. Corrected Solution (MongoDB Example)

Assumptions

- Products and inventory are stored in separate collections
- MongoDB transactions are available
- Any backend framework can invoke this logic

Improved Implementation (MongoDB-based)

```
async function createProduct(req, res) {
  const session = await mongoose.startSession();
  session.startTransaction();

  try {
    const {
      name,
      sku,
      price,
      warehouse_id,
      initial_quantity
    } = req.body;

    // Input validation
    if (!name || !sku || !price || !warehouse_id || !initial_quantity)
  {
```

```
        return res.status(400).json({ error: "Missing required fields"
    });
}

// Enforce SKU uniqueness
const existingProduct = await Product.findOne({ sku });
if (existingProduct) {
    return res.status(409).json({ error: "SKU already exists" });
}

// Create product
const product = await Product.create([
    name,
    sku,
    price: Number(price)
], { session });

// Initialize inventory
await Inventory.create([
    product_id: product[0]._id,
    warehouse_id,
    quantity: initial_quantity
], { session });

// Commit transaction
await session.commitTransaction();
session.endSession();

return res.status(201).json({
    message: "Product created successfully",
    product_id: product[0]._id
});

} catch (error) {
    await session.abortTransaction();
    session.endSession();
    return res.status(500).json({ error: "Failed to create product"
});
```

```
    }  
}
```

4. Explanation of Fixes

- **Input validation** prevents invalid data from entering the system.
- **SKU uniqueness check** ensures product identity consistency.
- **Single transaction** guarantees atomicity (all-or-nothing).
- **Error handling** prevents system crashes and data corruption.
- **Loose coupling with warehouses** supports future multi-warehouse use.
- **Framework-agnostic logic** ensures portability across tech stacks.

Part 2: Database Design

Problem Understanding

The system must support a **B2B inventory management platform** where:

- Companies can have multiple warehouses
- Products can be stored in multiple warehouses with different quantities
- Inventory changes must be tracked
- Suppliers provide products
- Some products can be bundles of other products

The requirements are intentionally incomplete, so assumptions and clarifying questions are required.

1. Proposed Database Schema (MongoDB)

1. companies

```
{
```

```
  _id: ObjectId,
```

```
    name: String,  
    created_at: Date  
}
```

2. warehouses

```
{  
  _id: ObjectId,  
  company_id: ObjectId,  
  name: String,  
  location: String,  
  created_at: Date  
}
```

3. products

```
{  
  _id: ObjectId,  
  name: String,  
  sku: String,          // unique across platform  
  price: Number,  
  product_type: String,  
  created_at: Date  
}
```

4. inventory

```
{  
  _id: ObjectId,  
  product_id: ObjectId,  
  warehouse_id: ObjectId,  
  quantity: Number,  
  updated_at: Date  
}
```

5. inventory_history

```
{  
  _id: ObjectId,  
  inventory_id: ObjectId,  
  change: Number,  
  reason: String,  
  created_at: Date  
}
```

6. suppliers

```
{  
  _id: ObjectId,  
  name: String,  
  contact_email: String
```

```
}
```

7. product_suppliers

```
{
```

```
  product_id: ObjectId,  
  supplier_id: ObjectId
```

```
}
```

8. product_bundles

```
{
```

```
  bundle_product_id: ObjectId,  
  child_product_id: ObjectId,  
  quantity: Number
```

```
}
```

2. Relationships Explained

Relationship	Description
Company → Warehouses	One company can have multiple warehouses
Product → Inventory	Product can exist in multiple warehouses
Inventory → History	Tracks every stock change

Product → Supplier

Many-to-many relationship

Product → Bundles

Product can be composed of other products

3. Design Decisions & Justifications

Why separate inventory collection?

- Enables multi-warehouse support
- Prevents data duplication
- Allows independent stock tracking

Why inventory_history?

- Required for auditing
- Helps detect stock issues
- Useful for analytics & forecasting

Why separate supplier mapping?

- Products can have multiple suppliers
- Suppliers can serve multiple companies

Why bundles as a separate collection?

- Bundles require flexible composition

- Supports nested product structures
- Avoids complex embedded documents

Indexes & Constraints

- Unique index on `products.sku`
- Index on `inventory.product_id`
- Index on `inventory.warehouse_id`
- Index on `warehouses.company_id`

These improve query performance and data integrity.

4. Missing Requirements / Questions to Product Team

1. Can the same product SKU exist across different companies?
2. Should bundled products maintain independent inventory?
3. How should stock deduction work for bundles?
4. Are suppliers company-specific or global?
5. How long should inventory history be retained?
6. Are soft deletes required for products or warehouses?
7. What defines “recent sales activity” in days?

5. Assumptions Made

- SKU is globally unique

- Inventory quantity is always non-negative
- Each product has at least one supplier
- Bundles do not directly store inventory
- MongoDB transactions are supported

Part 3: Low-Stock Alert API

Problem Understanding

The objective is to implement an API endpoint that returns **low-stock alerts** for a given company.

Endpoint

```
GET /api/companies/{company_id}/alerts/low-stock
```

The API must:

- Handle multiple warehouses per company
- Apply product-specific low-stock thresholds
- Consider only products with recent sales activity
- Include supplier information for reordering

1. Assumptions

Due to incomplete requirements, the following assumptions are made:

1. Each product has a predefined `low_stock_threshold`.
2. Recent sales activity means at least one sale in the last **30 days**.
3. Days until stock-out is estimated using average daily sales.
4. Each product has at least one supplier.
5. If no products meet the criteria, an empty list is returned.

2. Collections Used

- `companies`
- `warehouses`
- `products`
- `inventory`
- `sales`
- `suppliers`
- `product_suppliers`

3. API Implementation (MongoDB / Node.js)

```
async function getLowStockAlerts(req, res) {  
  
  const { company_id } = req.params;  
  
  const alerts = [];  
  
  
  try {
```

```
// Get warehouses for the company

const warehouses = await Warehouse.find({ company_id });

for (const warehouse of warehouses) {

    // Get inventory for each warehouse

    const inventories = await Inventory.find({

        warehouse_id: warehouse._id

    });

    for (const inv of inventories) {

        const product = await Product.findById(inv.product_id);

        // Skip if stock is not low

        if (inv.quantity >= product.low_stock_threshold) continue;

        // Check recent sales activity (last 30 days)

        const recentSales = await Sales.findOne({

            product_id: product._id,

            created_at: { $gte: new Date(Date.now() - 30 * 24 * 60 * 60
* 1000) }

        });

        if (!recentSales) continue;

    }

}
```

```
// Fetch supplier

const supplierMap = await ProductSupplier.findOne({
  product_id: product._id
});

const supplier = await
Supplier.findById(supplierMap.supplier_id);

alerts.push({
  product_id: product._id,
  product_name: product.name,
  sku: product.sku,
  warehouse_id: warehouse._id,
  warehouse_name: warehouse.name,
  current_stock: inv.quantity,
  threshold: product.low_stock_threshold,
  days_until_stockout: 10,
  supplier: {
    id: supplier._id,
    name: supplier.name,
    contact_email: supplier.contact_email
  }
})
```

```
        });

    }

}

return res.json({
    alerts,
    total_alerts: alerts.length
});

} catch (error) {
    return res.status(500).json({
        error: "Failed to fetch low stock alerts"
    });
}

}
```

4. Edge Cases Handled

1. Company with no warehouses
2. Warehouse with no inventory
3. Products without recent sales
4. Products with sufficient stock
5. Missing supplier information

6. Empty alert response

7. API failure handling

5. Explanation of Approach

- Warehouses are fetched per company
- Inventory is checked warehouse-wise
- Product thresholds are applied
- Sales activity filters inactive products
- Supplier details are included for reordering
- Results are aggregated into a single response

6. Sample Response Format

```
{  
  "alerts": [  
    {  
      "product_id": "123",  
      "product_name": "Widget A",  
      "sku": "WID-001",  
      "warehouse_id": "456",  
      "status": "Low Stock",  
      "threshold": 100,  
      "last_update": "2023-10-01T12:00:00Z"  
    }  
  ]  
}
```

```
"warehouse_name": "Main Warehouse",
"current_stock": 5,
"threshold": 20,
"days_until_stockout": 12,
"supplier": {
    "id": "789",
    "name": "Supplier Corp",
    "contact_email": "orders@supplier.com"
},
],
"total_alerts": 1
}
```