Inventory Management System for B2B SaaS

Part 1: Code Review & Debugging

Issues Identified

After reviewing the code, I found several problems that would definately cause issues in production:

1. No Input Validation

The code directly accesses dictionary keys without checking if they exist. If a client sends incomplete data, this will crash with a KeyError.

Impact: The API will return 500 errors instead of helpful validation messages. Users wont know what they did wrong.

2. Missing Error Handling

There's no try-catch block around database operations. If anything fails during commit, the application crashes.

Impact: Partial data might be saved (product created but not inventory), leaving the databse in an inconsistant state.

3. No SKU Uniqueness Check

The requirements say SKUs must be unique, but the code doesn't check for duplicates before inserting.

Impact: Either database constraint violation (if we have a unique index) or duplicate SKUs in the system (if we don't), both are bad scenarios.

4. Redundant warehouse_id

We're storing warehouse_id in both the Product and Inventory tables. This doesn't make sense because products can exist in multiple warehouses according to the requirements.

Impact: The data model is fundamentaly broken. If a product exists in 3 warehouses, how do we decide which warehouse_id goes in the Product table?

5. Transaction Issues

Two separate commits mean if the second one fails, we have a product without inventory. Should be a single transaction.

Inventory Management System for B2B SaaS

Impact: Data inconsitency - products exist without corresponding inventory records.

6. No Authentication/Authorization

No check to see if the user has permission to create products or access this warehouse.

Impact: Security vulnerability - anyone can create products in any warehouse.

Corrected Version (Node.js/Express + MongoDB)

```
// routes/products.js

const express = require('express');

const router = express.Router();

const Product = require('../models/Product');

const Inventory = require('../models/Inventory');

const Warehouse = require('../models/Warehouse');

const mongoose = require('mongoose');


// Simple auth middleware (inline for this example)

const authenticateUser = async (req, res, next) => {

  // In real app, verify JWT token here

  // For now, assuming user object is attached to request

  if (!req.user || !req.user.companyId) {

    return res.status(401).json({ error: 'Authentication required' });

  }

  next();
```

Inventory Management System for B2B SaaS

```
};


router.post('/api/products', authenticateUser, async (req, res) => {
  // Start a session for transaction
  const session = await mongoose.startSession();
  session.startTransaction();

  try {
    const { name, sku, price, warehouseId, initialQuantity } = req.body;

    // Validate required fields
    const requiredFeilds = ['name', 'sku', 'price', 'warehouseId', 'initialQuantity'];
    const missingFields = [];

    if (!name) missingFields.push('name');
    if (!sku) missingFields.push('sku');
    if (price === undefined) missingFields.push('price');
    if (!warehouseId) missingFields.push('warehouseId');
    if (initialQuantity === undefined) missingFields.push('initialQuantity');

    if (missingFields.length > 0) {
```

```
  await session.abortTransaction();
  return res.status(400).json({
    error: 'Missing required fields',
    fields: missingFields
  });
}

// Validate data types and ranges
const priceNum = parseFloat(price);
if (isNaN(priceNum) || priceNum < 0) {
  await session.abortTransaction();
  return res.status(400).json({
    error: 'Price must be a positive number'
  });
}

const quantityNum = parseInt(initialQuantity);
if (isNaN(quantityNum) || quantityNum < 0) {
  await session.abortTransaction();
  return res.status(400).json({
    error: 'Quantity must be a positive integer'
```

```
  });
}

// Check if SKU already exists for this company
const existingProd = await Product.findOne({
  companyId: req.user.companyId,
  sku: sku.trim().toUpperCase()
});

if (existingProd) {
  await session.abortTransaction();
  return res.status(409).json({
    error: 'SKU already exists',
    existingProductId: existingProd._id
  });
}

// Verify warehouse exists and user has access
const warehouse = await Warehouse.findOne({
  _id: warehouseId,
  companyId: req.user.companyId,
```

Inventory Management System for B2B SaaS

```
  isActive: true
});

if (!warehouse) {
  await session.abortTransaction();
  return res.status(404).json({
    error: 'Warehouse not found or inactive'
  });
}

// Create product (without warehouseId in the product model)
const newProduct = new Product({
  companyId: req.user.companyId,
  name: name.trim(),
  sku: sku.trim().toUpperCase(),
  price: priceNum,
  createdBy: req.user._id,
  isActive: true
});

await newProduct.save({ session });
```

Inventory Management System for B2B SaaS

```
// Create inventory record
const newInventory = new Inventory({
  productId: newProduct._id,
  warehouseId: warehouseId,
  quantity: quantityNum,
  companyId: req.user.companyId
});

await newInventory.save({ session });

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

return res.status(201).json({
  message: 'Product created succesfully',
  productId: newProduct._id,
  sku: newProduct.sku,
  initialInventory: {
    warehouseId: warehouse._id,
```

```
        warehouseName: warehouse.name,
        quantity: quantityNum
      }
    });

  } catch (err) {
    await session.abortTransaction();
    session.endSession();

    console.error('Error creating product:', err);

    return res.status(500).json({
      error: 'An error occured while creating the product'
    });
  }
});

module.exports = router;
```

Part 2: Database Design

Inventory Management System for B2B SaaS

Here's my proposed schema using MongoDB since we're using MERN stack:

MongoDB Schema Design

```
// models/Company.js
const mongoose = require('mongoose');

const companySchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    trim: true
  },
  isActive: {
    type: Boolean,
    default: true
  },
  createdAt: {
    type: Date,
    default: Date.now
  },
  updatedAt: {
    type: Date,
    default: Date.now
  }
});

module.exports = mongoose.model('Company', companySchema);
```

Design Decisions & Justifications:

1. Company Isolation Every document has companyId indexed. This is critical for multi-tenant SaaS - we cant have Company A seeing Company B's data. Also helps with data partitioning as the system scales.

2. SKU Uniqueness Created compound index on (companyId, sku) instead of globally unique. Different companies might use the same SKU systems, and we dont want collisions.

3. Embedded vs Referenced Used embedded documents for bundle items and suppliers within Product since they're tightly coupled. But kept Inventory separate since it changes frequently.

Inventory Management System for B2B SaaS

4. Soft Deletes Added isActive flags instead of hard deletes. In a B2B system, you almost never want to actualy delete data - historical reports need it.

5. Indexes Added compound indexes on frequently queried fields and foreign key references. MongoDB benefits from good indexing strategy.

6. Denormalization Stored companyId in multiple collections for faster queries without joins. MongoDB works better with some denormalization.

Part 3: API Implementation

Assumptions I'm Making:

1. "Recent sales activity" means sales in the last 30 days
2. "Days until stockout" calculated as: current_stock / avg_daily_sales
3. Low stock threshold from product type, or default of 10 if not set
4. Only alert if current stock is below threshold
5. Use primary supplier for reorder info
6. Exclude deleted/inactive products/warehouses

Implementation (Node.js/Express + MongoDB):

```
// routes/alerts.js
const express = require('express');
const router = express.Router();
const mongoose = require('mongoose');
const Product = require('../models/Product');
const Inventory = require('../models/Inventory');
const Warehouse = require('../models/Warehouse');
const ProductType = require('../models/ProductType');
const Sale = require('../models/Sale');
const Supplier = require('../models/Supplier');

// Simple auth middleware
const authenticateUser = async (req, res, next) => {
  if (!req.user || !req.user.companyId) {
    return res.status(401).json({ error: 'Authentication
required' });
  }
  next();
};

/**
```

Inventory Management System for B2B SaaS

```
 * GET /api/companies/:companyId/alerts/low-stock
 * Returns low stock alerts for products with recent sales activity
 *
 * Query params:
 * - days: Number of days to look back for sales (default 30)
 * - warehouseId: Optional filter by specific warehouse
 */
router.get('/api/companies/:companyId/alerts/low-stock',
  authenticateUser,
  async (req, res) => {

  try {
    const { companyId } = req.params;

    // Verify user has access to this company
    if (req.user.companyId.toString() !== companyId) {
      return res.status(403).json({
        error: 'Unauthorized access to company data'
      });
    }

    // Get query parameters
    const lookbackDays = parseInt(req.query.days) || 30;
    const warehouseFilter = req.query.warehouseId;

    // Validate lookback days
    if (lookbackDays < 1 || lookbackDays > 365) {
      return res.status(400).json({
        error: 'Days parameter must be between 1 and 365'
      });
    }

    const cutoffDate = new Date();
    cutoffDate.setDate(cutoffDate.getDate() - lookbackDays);

    // Build aggregation pipeline
    const matchStage = {
      companyId: mongoose.Types.ObjectId(companyId),
      isActive: true
    };
```

Inventory Management System for B2B SaaS

```
    if (warehouseFilter) {
      matchStage.warehouseId =
mongoose.Types.ObjectId(warehouseFilter);
    }

    // Get inventory items with low stock
    const inventoryItems = await Inventory.aggregate([
      {
        $match: matchStage
      },
      {
        $lookup: {
          from: 'products',
          localField: 'productId',
          foreignField: '_id',
          as: 'product'
        }
      },
      {
        $unwind: '$product'
      },
      {
        $match: {
          'product.isActive': true
        }
      },
      {
        $lookup: {
          from: 'warehouses',
          localField: 'warehouseId',
          foreignField: '_id',
          as: 'warehouse'
        }
      },
      {
        $unwind: '$warehouse'
      },
      {
        $match: {
          'warehouse.isActive': true
```

Inventory Management System for B2B SaaS

```
    }
  },
  {
    $lookup: {
      from: 'producttypes',
      localField: 'product.productTypeId',
      foreignField: '_id',
      as: 'productType'
    }
  },
  {
    $lookup: {
      from: 'sales',
      let: {
        prodId: '$productId',
        whId: '$warehouseId'
      },
      pipeline: [
        {
          $match: {
            $expr: {
              $and: [
                { $eq: ['$productId', '$$prodId'] },
                { $eq: ['$warehouseId', '$$whId'] },
                { $gte: ['$saleDate', cutoffDate] }
              ]
            }
          }
        },
        {
          $group: {
            _id: null,
            totalSales: { $sum: '$quantity' }
          }
        }
      ],
      as: 'salesData'
    }
  },
  {
    $project: {
```

Inventory Management System for B2B SaaS

```
          productId: '$product._id',
          productName: '$product.name',
          sku: '$product.sku',
          warehouseId: '$warehouse._id',
          warehouseName: '$warehouse.name',
          currentStock: '$quantity',
          lowStockThreshold: '$lowStockThreshold',
          defaultThreshold: {
            $ifNull: [
              { $arrayElemAt:
['$productType.defaultLowStockThreshold', 0] },
              10
            ]
          },
          suppliers: '$product.suppliers',
          totalSales: {
            $ifNull: [
              { $arrayElemAt: ['$salesData.totalSales', 0] },
              0
            ]
          }
        }
      }
    ]);

    const alerts = [];

    // Process each inventory item
    for (const item of inventoryItems) {
      // Determine threshold
      const threshold = item.lowStockThreshold ||
item.defaultThreshold;

      // Calculate average daily sales
      const avgDailySales = item.totalSales / lookbackDays;

      // Only include if below threshold AND has recent sales
      if (item.currentStock < threshold && avgDailySales > 0)
{
        // Calculate days until stockout
        let daysUntilStockout = null;
```

Inventory Management System for B2B SaaS

```
        if (avgDailySales > 0) {
          daysUntilStockout = Math.floor(item.currentStock /
avgDailySales);
        }

        // Find primary supplier
        let supplierInfo = null;
        if (item.suppliers && item.suppliers.length > 0) {
          const primarySupplier = item.suppliers.find(s =>
s.isPrimary);
          const supplierToUse = primarySupplier ||
item.suppliers[0];

          if (supplierToUse) {
            // Fetch full supplier details
            const supplier = await Supplier.findOne({
              _id: supplierToUse.supplierId,
              isActive: true
            });

            if (supplier) {
              supplierInfo = {
                id: supplier._id,
                name: supplier.name,
                contactEmail: supplier.contactEmail,
                leadTimeDays: supplierToUse.leadTimeDays
              };
            }
          }
        }

        alerts.push({
          productId: item.productId,
          productName: item.productName,
          sku: item.sku,
          warehouseId: item.warehouseId,
          warehouseName: item.warehouseName,
          currentStock: item.currentStock,
          threshold: threshold,
          daysUntilStockout: daysUntilStockout,
```

Inventory Management System for B2B SaaS

```
          avgDailySales:
parseFloat(avgDailySales.toFixed(2)),
          supplier: supplierInfo
        });
      }
    }

    // Sort by urgency (lowest days first)
    alerts.sort((a, b) => {
      if (a.daysUntilStockout === null) return 1;
      if (b.daysUntilStockout === null) return -1;
      return a.daysUntilStockout - b.daysUntilStockout;
    });

    return res.status(200).json({
      alerts: alerts,
      totalAlerts: alerts.length,
      parameters: {
        lookbackDays: lookbackDays,
        warehouseId: warehouseFilter || null
      }
    });

  } catch (err) {
    console.error('Error fetching low stock alerts:', err);

    return res.status(500).json({
      error: 'An error occured while fetching alerts'
    });
  }
});

module.exports = router;
```

Edge Cases Handled:

1. **No sales data**: Products without sales wont appear in alerts (no "recent activity")
2. **Division by zero**: If avgDailySales is 0, we handle it gracefuly
3. **Missing supplier**: Not all products have suppliers - we return null
4. **Multiple suppliers**: We use the primary supplier, or first one if no primary
5. **Inactive records**: Filter out inactive products/warehouses/suppliers
6. **Authorization**: Check user belongs to the company
7. **Invalid parameters**: Validate the lookback days range

Inventory Management System for B2B SaaS

8. **Empty results**: Returns empty array with totalAlerts: 0

Performance Considerations:

Potential Issues:

- Aggregation pipeline could get slow with millions of products/sales
- The lookup for sales data is expensive
- Fetching supplier details in a loop (N+1 problem)

Optimizations I'd Consider:

1. Cache results for 1 hour since stock doesnt change rapidly
2. Add pagination with limit/skip for companies with many products
3. Pre-calculate avgDailySales nightly and store in inventory document
4. Use Redis for caching frequently accessed data
5. Fetch all suppliers in one query instead of in loop

Summary & Reflection

What Went Well:

- Identified the core issue in Part 1 (warehouse_id in wrong table)
- Comprehensive database design with good normalization
- Handled authentication and authorization properly
- Thought about multi-tenancy from the start

What I Struggled With:

- Part 3 took longer than expected because I kept second-guessing the sales velocity calculation
- Not 100% sure about the bundle products implementation - might need a recursive query for nested bundles
- The supplier query in Part 3 has N+1 issue that I noted but didn't fix (ran out of time)

Assumptions I Made:

- Using MongoDB for MERN stack
- SKUs are unique per company, not globally
- Prices don't vary by customer or warehouse
- "Recent activity" means last 30 days
- Single currency for all prices
- Products are measured in simple units (not weight/volume)