Title: StockFlow – Inventory Management System (B2B SaaS)

1. Assumptions

Since requirements were intentionally incomplete, I made the following assumptions:

- SKU is globally unique across the platform
- Recent sales activity means at least one sale in last 30 days
- One primary supplier per product
- Inventory quantity cannot be negative
- Price is stored as DECIMAL for financial accuracy

Part 1: Code Review & Debugging (30 minutes)

```
// routes/products.js
const express = require("express");
const router = express.Router();
const { Product, Inventory, Warehouse, sequelize } = require("../models");

router.post("/api/products", async (req, res) => {
  const {
    name,
    sku,
    price,
    warehouse_id,
    initial_quantity = 0
  } = req.body;

  // Basic validation
  if (!name || !sku || price == null) {
    return res.status(400).json({
      error: "name, sku and price are required"
    });
  }

  const transaction = await sequelize.transaction();

  try {
```

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

// Validate warehouse (if provided)
if (warehouse_id) {
  const warehouse = await Warehouse.findByPk(warehouse_id);
  if (!warehouse) {
    await transaction.rollback();
    return res.status(404).json({
      error: "Warehouse not found"
    });
  }
}

// Create product (product is warehouse-independent)
const product = await Product.create(
  {
    name,
    sku,
    price: parseFloat(price) // handled as DECIMAL in DB
  },
  { transaction }
);

// Create inventory only if warehouse provided
if (warehouse_id) {
  await Inventory.create(
    {
      product_id: product.id,
      warehouse_id,
      quantity: initial_quantity
    },
    { transaction }
  );
}

// Commit transaction
await transaction.commit();

return res.status(201).json({
  message: "Product created successfully",
  product_id: product.id
```

```
    });

  } catch (error) {
    // Rollback on failure
    await transaction.rollback();
    console.error(error);

    return res.status(500).json({
      error: "Internal server error"
    });
  }
});

module.exports = router;
```

Code Review & Debugging – Justification

- Validation is required to prevent crashes when input data is missing or invalid.
- SKU uniqueness must be enforced to avoid duplicate products.
- Database transactions are used to prevent partial data creation.
- Product and inventory are separated to support multiple warehouses.
- Decimal price handling avoids financial calculation errors.

Why this matters:
It ensures data consistency, business rule enforcement, and production stability.

Part 2: Database Design (25 minutes)

Tables & Relationships

```
companies (
  id             UUID PRIMARY KEY,
  name           VARCHAR(255) NOT NULL,
  created_at     TIMESTAMP DEFAULT NOW()
)

warehouses (
  id             UUID PRIMARY KEY,
  company_id      UUID REFERENCES companies(id),
  name            VARCHAR(255) NOT NULL,
  location        TEXT,
  created_at      TIMESTAMP DEFAULT NOW()
)

products (
  id              UUID PRIMARY KEY,
  company_id      UUID REFERENCES companies(id),
  name            VARCHAR(255) NOT NULL,
  sku             VARCHAR(100) NOT NULL UNIQUE,
  price           DECIMAL(10,2) NOT NULL,
  product_type    VARCHAR(50), -- simple, bundle, raw
  created_at      TIMESTAMP DEFAULT NOW()
)

inventory (
  id              UUID PRIMARY KEY,
  product_id      UUID REFERENCES products(id),
  warehouse_id    UUID REFERENCES warehouses(id),
  quantity        INTEGER NOT NULL DEFAULT 0,
  updated_at      TIMESTAMP DEFAULT NOW(),
  UNIQUE (product_id, warehouse_id)
)

inventory_transactions (
  id              UUID PRIMARY KEY,
  product_id      UUID REFERENCES products(id),
  warehouse_id    UUID REFERENCES warehouses(id),
  change_quantity INTEGER NOT NULL,
  reason          VARCHAR(50), -- sale, restock, adjustment
  created_at      TIMESTAMP DEFAULT NOW()
)

suppliers (
  id              UUID PRIMARY KEY,
  name            VARCHAR(255) NOT NULL,
  contact_email   VARCHAR(255),
  phone           VARCHAR(50),
  created_at      TIMESTAMP DEFAULT NOW()
)

product_suppliers (
  product_id      UUID REFERENCES products(id),
  supplier_id     UUID REFERENCES suppliers(id),
  lead_time_days  INTEGER,
  PRIMARY KEY (product_id, supplier_id)
)
```

```
product_bundles (
  bundle_id        UUID REFERENCES products(id),
  child_product_id UUID REFERENCES products(id),
  quantity         INTEGER NOT NULL,
  PRIMARY KEY (bundle_id, child_product_id)
)

product_thresholds (
  product_id       UUID REFERENCES products(id),
  threshold        INTEGER NOT NULL
)
```

Database Design – Justification

- Separate tables for products, warehouses, and inventory allow multi-warehouse support.
- Inventory transaction history enables auditing and stock tracking.
- Many-to-many relationships support multiple suppliers and bundles.
- Proper constraints and indexes improve data integrity and query performance.

Why this matters:
The design is scalable, flexible, and supports real-world inventory operations.

Part 3: API Implementation (35 minutes)

```
const express = require("express");

const router = express.Router();

const { Op } = require("sequelize");



const {

  Company,

  Warehouse,
  Product,
  Inventory,
  ProductThreshold,
  InventoryTransaction,
  Supplier,
  ProductSupplier
} = require("../models");

router.get("/api/companies/:company_id/alerts/low-stock", async (req, res) => {
  const { company_id } = req.params;

  try {
    // Fetch warehouses of company
    const warehouses = await Warehouse.findAll({
      where: { company_id }
    });

    if (!warehouses.length) {
      return res.json({ alerts: [], total_alerts: 0 });
    }

    const warehouseIds = warehouses.map(w => w.id);
    const thirtyDaysAgo = new Date(Date.now() - 30 * 24 * 60 * 60 * 1000);

    // Fetch inventory with product + threshold
    const inventories = await Inventory.findAll({
```

```
  where: { warehouse_id: warehouseIds },
  include: [
    {
      model: Product,
      include: [ProductThreshold]
    }
  ]
});

const alerts = [];

for (const item of inventories) {
  const product = item.Product;
  const thresholdRow = product.ProductThreshold;

  // Skip if no threshold
  if (!thresholdRow) continue;

  // Check recent sales activity
  const sales = await InventoryTransaction.findAll({
    where: {
      product_id: product.id,
      warehouse_id: item.warehouse_id,
      reason: "sale",
      created_at: { [Op.gte]: thirtyDaysAgo }
    }
});

if (!sales.length) continue;

// Calculate average daily sales
const totalSold = sales.reduce(
  (sum, tx) => sum + Math.abs(tx.change_quantity),
  0
);
const avgDailySales = totalSold / 30;

// Avoid divide by zero
const daysUntilStockout =
  avgDailySales > 0
    ? Math.floor(item.quantity / avgDailySales)
    : null;

// Check low stock condition
if (item.quantity >= thresholdRow.threshold) continue;

// Fetch supplier info
const productSupplier = await ProductSupplier.findOne({
  where: { product_id: product.id },
  include: [Supplier]
});
```

```
  const warehouse = warehouses.find(w => w.id === item.warehouse_id);

  alerts.push({
    product_id: product.id,
    product_name: product.name,
    sku: product.sku,
    warehouse_id: item.warehouse_id,
    warehouse_name: warehouse.name,
    current_stock: item.quantity,
    threshold: thresholdRow.threshold,
    days_until_stockout: daysUntilStockout,
    supplier: productSupplier
      ? {
          id: productSupplier.Supplier.id,
          name: productSupplier.Supplier.name,
          contact_email: productSupplier.Supplier.contact_email
        }
      : null
  });
}

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

```
  } catch (error) {
    console.error(error);
    return res.status(500).json({
      error: "Internal server error"
    });
  }
});
module.exports = router;
```

API Implementation – Justification

- Alerts are generated per warehouse to give accurate stock visibility.
- Thresholds are product-specific to match business needs.
- Recent sales activity is checked to avoid unnecessary alerts.
- Supplier details are included to enable quick reordering.

Why this matters:
It provides actionable, accurate alerts that help businesses prevent stock-outs.