Inventory Management System for B2B SaaS

Part - 1

Following things are considered while debugging the API

1. The Product and Inventory relations contain all the columns used in API calls.
2. There are no other not-null columns in relation.

Issues in current code:

1. Unnecessary warehouse_id field in product.
    a. A product is a catalog entry i.e. it is global for all Inventory with unique SKUs.
    b. Inventory should tie a product to a warehouse.
    c. Impact: Having warehouse in product tables, seems that product exists only for a single warehouse, which breaks ‘product can exist for multiple ware house’ constraint.
    d. Fix:
        i. Remove warehouse_id from product
        ii. Use warehouse_id, product_id pairs to find product availability in different warehouses.

2. No default value assignment if value is not present
    a. For optional fields default value is not assigned if the field is not present.
    b. Impact: Client sends partial data - server crash with KeyError.
    c. Fix:
        i. Use .get() with defaults
        ii. Example: data.get(‘initial_quantity’, 0)

3. Data received from the request body is not validated on server-side.
    a. Data received via request body is not validated which might lead to in-consistent data.
    b. Example:
        i. For Product price:
        Negative price or string should not be accepted.
        ii. For Inventory initial_quantity**:**
        Very large numbers such as 1000000 and negative numbers should not be accepted.
    c. **Impact:** Leads to inconsistent / corrupted business data.
    d. Fix:
        i. Checking if all the required keys are present or not.
        ii. Validating all the data types.
        iii. Use a schema validation library for validations.
        iv. If received data is not valid, return error messages on which fields are not valid.

4. Multiple commits under a single transaction
    a. Product and Inventory are committed separately.
    b. If Product commit succeeds but inventory commit fails DB is left in inconsistent state.
    c. Product exists but its Inventory is not present.
    d. Impact: Broken stock management.
    e. Fix:
        i. Instead of committing, flush the changes so that data can be stored into DB and auto-generated keys can be utilized further.
        ii. Once all the DB operations are completed without any errors, commit all the changes once.
        iii. If any error occurs, rollback the changes.

5. No rollback on errors.
    a. If any errors occur during insertion of product or inventory DB is left in an inconsistent state without rollback.
    b. Example: If product is inserted successfully, but inventory fails, then all the previous transactions before inventory must be rolled-back.
    c. Impact: Block Further transactions or leaves the DB half-written.
    d. Fix:
        i. Handle errors using try-catch block
        ii. Following are the errors that may occur during insertion
            1. **IntegrityError**: If SKU is duplicate
            2. **InternalServerError:** Any other uncertain error.
        iii. If any error occurs, rollback all the changes and send appropriate error messages to the client.
        iv. Possible Error messages are:
            1. **IntegrityError**: 409 - Conflict
            2. **InternalServerError:** 500 - Internal Server Error

6. SKU must be unique
    a. No unique check for SKU's uniqueness.
    b. Impact: Impossible to reliably identify products across inventory, sales, and external integrations.
    c. Fix:
        i. Use unique-key constraints in DB. It will throw an IntegrityError if SKU already exists.
        ii. If schema can't be updated - Search for SKU name in DB if present return error response with status code 409, if not present continue with further operations.

7. No HTTP status code
    a. Always returns only a single message and product Id after product creation, without HTTP code 201-created
    b. Without HTTP code clients cannot distinguish between success and failure easily.
    c. Impact: Client can't distinguish between :
        i. Successful Creation (201 Created)
        ii. Bad Input (400 Bad Request)
        iii. Conflict (409 Conflict)
        iv. Server error (500 Internal Server Error)
    d. Fix: Return appropriate status codes with response.

8. Not closing resources properly after use.
    a. db.session() needs to be closed after the transaction is completed successfully.
    b. Even though sessions have request level scope, still it is good practice to close them explicitly.
    c. **Impact:** Session left in dirty state, may block future operations memory leaks or connection pools exhaustion.
    d. Fix: Use finally block to close the DB session.

9. Checking warehouse with warehouse_id exists before insertion
    a. Impact:
        i. Leads to inconsistent stock management.
        ii. It might insert record for warehouse which might not exists
    b. Fix: Ensure foreign key constraint is applied properly for warehouse_id in inventory table.
    c. Else manually check whether the warehouse exists or not.

10. Insertion of duplicate products.
    a. This code allows the blind insertion of the products before checking whether the SKU already exists in the database.
    b. **Impact:** The same product may be inserted multiple times with different Ids, breaking the uniqueness of the catalog.
    c. Fix:
        i. Ensure uniqueness of SKU before product insertion.

11. Always inventory is created with ‘initial_quantity’
    a. Even if the inventory exists for some (product, warehouse), the code inserts new inventory instead of updating.
    b. Impact: Multiple inventory records for same product-warehouse.
    c. Fix:
        i. Before product insertion check whether products already exist by SKU.
        ii. If a product exists, then get the product by SKU.
        iii. Using product_id and warehouse_id check if the inventory is already present or not.
        iv. If yes, update the inventory, else create the inventory.

Update 2) Though in RESTful APIs, each end-point should be responsible for only a single operation, i.e. either insertion of inventory or updation of inventory.
So here the updation of inventory could be kept optional, i.e. if the product exists with SKU, we can directly return 409 - conflict, product with SKU already exists.

Update 3) The better we can do it, we can separate both the operations with separate end-points.
One for product creation, and one for inventory creation / updation.
If inventory is already present, it will be updated, if not present it will be created

1) Updated Code - Assuming Inventory should be updated if product exists

```
# Import required libraries
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError
from app import app, db
from models import Product, Inventory, Warehouse

product_schema = AddProductSchema()
@app.route('/api/products', methods=['POST'])
def create_product():

# Validate the AddProductSchema
# If error input is invalid then return the error
# Else process the input
    try:
        data = product_schema.load(request.json or {})
    except ValidationError as err:
        return jsonify({"errors": err.messages}), 400

    try:
    # Check if warehouse with warehouse_id exists
    # If not exists return error response
    # Else process the input
        warehouse = db.session.get(Warehouse, data["warehouse_id"])
        if not warehouse:
            return jsonify({"error": "Warehouse not found"}), 404

    # Check if Product already exists by SKU
    # If not, add the product into the database and continue further process.
        product = Product.query.filter_by(sku=data["sku"]).first()
        if not product:
            product = Product(
                name=data["name"],
                sku=data["sku"],
                price=data["price"]
            )
            db.session.add(product)
            db.session.flush()

    # Check if inventory exist for the the product
    # If yes, update the inventory
    # Else create the inventory
    # As this was the last operaion commit the changes
        inventory = Inventory.query.filter_by(
```

```
            product_id=product.id,
            warehouse_id=data["warehouse_id"]
        ).first()

        if inventory:
            inventory.quantity += data["initial_quantity"]
        else:
            inventory = Inventory(
                product_id=product.id,
                warehouse_id=data["warehouse_id"],
                quantity=data["initial_quantity"]
            )
            db.session.add(inventory)

        db.session.commit()

    # After successful insertion return a success result
        return jsonify({
            "status": 201,
            "message": "Product created/updated successfully"
        }), 201


    # As we're checking in start only whether SKU is present or not, then only we're performing our next operations
    # Considering multiple request hit API concurrently with same SKU and both found that SKU is not present
    # At that time, one will be inserted successfully, while other will get an IntegrityError
    # And rollback all the previous operations

    except IntegrityError:
        db.session.rollback()
        return jsonify({"error": "SKU must be unique"}), 409

    # If any other exception occurs unexpectedly, like server crash, power failure, rollback the changes to maintain the DB consistency
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": f"Internal Server Error: {str(e)}"}), 500

    # Finally when all the operations are performed Close the DB connection
    finally:
        db.session.close()
```

2) Assuming if Product exists return 409 - conflict - Product with SKU already exists.

```
# Import required libraries
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError
from app import app, db
from models import Product, Inventory, Warehouse

product_schema = AddProductSchema()
@app.route('/api/products', methods=['POST'])
def create_product():

# Validate the AddProductSchema
# If error input is invalid then return the error
# Else process the input
    try:
        data = product_schema.load(request.json or {})
    except ValidationError as err:
        return jsonify({"errors": err.messages}), 400

    try:
    # Check if warehouse with warehouse_id exists
    # If not exists return error response
    # Else process the input
        warehouse = db.session.get(Warehouse, data["warehouse_id"])
        if not warehouse:
            return jsonify({"error": "Warehouse not found"}), 404

# Directly insert the product into DB, if SKU is present then, IntegrityError will be
thrown, which is handled in the except block
        product = Product(
                name=data["name"],
                sku=data["sku"],
                price=data["price"]
        )
        db.session.add(product)
        db.session.flush()
```

```
    # Check if inventory exist for the the product
    # If yes, update the inventory
    # Else create the inventory
    # As this was the last operation commit the changes
    # This code will be executed only and only if, new product is created, so inventory will always be None here. So we can easily remove this code.
        inventory = Inventory.query.filter_by(
            product_id=product.id,
            warehouse_id=data["warehouse_id"]
        ).first()

        if inventory:
            inventory.quantity += data["initial_quantity"]
        else:

            inventory = Inventory(
                product_id=product.id,
                warehouse_id=data["warehouse_id"],
                quantity=data["initial_quantity"]
            )
            db.session.add(inventory)

            db.session.commit()

    # After successful insertion return a success result
        return jsonify({
            "status": 201,
            "message": "Product created/updated successfully"
        }), 201

    # As we're checking in start only whether SKU is present or not, then only we're performing our next operations
    # Considering multiple request hit API concurrently with same SKU and both found that SKU is not present
    # At that time, one will be inserted successfully, while other will get an IntegrityError
    # And rollback all the previous operations

    except IntegrityError:
        db.session.rollback()
        return jsonify({"error": "SKU must be unique"}), 409

    # If any other exception occurs unexpectedly, like server crash, power failure, rollback the changes to maintain the DB consistency
```

```
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": f"Internal Server Error: {str(e)}"}), 500

    # Finally when all the operations are performed Close the DB connection
    finally:
        db.session.close()
```

AddProductSchema

This class is used for validation purpose, it will validate all the fields related to Product relation.

```
from marshmallow import Schema, fields, validate, ValidationError
class ProductSchema(Schema):

# Length of the name should be between 1 to 255
    name = fields.String(required=True, validate=validate.Length(min=1, max=255))

# Length of the name should be between 1 to 100
    sku = fields.String(required=True, validate=validate.Length(min=1, max=100))

# Min price for a product should be 0
# Price will be stored as Decimal with 2 precision points
    price = fields.Decimal(required=True, places=2, validate=validate.Range(min=0))

# Warehouse_id is required
    warehouse_id = fields.Integer(required=True)

# It is optional with range of 0 - 1000000
# If not present sets default value as 0
    initial_quantity = fields.Integer(
        missing=0,  # default if not provided
        validate=validate.Range(min=0, max=1000000)
    )
```

3) Separate end-points for both product and inventory.

```
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError
from app import app, db
from models import Product, Inventory, Warehouse
from schemas import AddProductSchema, AddInventorySchema
from marshmallow import ValidationError

product_schema = AddProductSchema()
inventory_schema = AddInventorySchema()


@app.route('/api/products', methods=['POST'])
def create_product():
    try:
        data = product_schema.load(request.json or {})
    except ValidationError as err:
        return jsonify({"errors": err.messages}), 400

    try:
        # If product already exist throw 409 - conflict erorr
        existing = Product.query.filter_by(sku=data["sku"]).first()
        if existing:
            return jsonify({
                "error": "Product with this SKU already exists"
            }), 409

        # Create new product
        product = Product(
            name=data["name"],
            sku=data["sku"],
            price=data["price"],
            description=data.get("description")
        )
        db.session.add(product)
        db.session.commit()

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

```
    except IntegrityError:
        db.session.rollback()
        return jsonify({"error": "SKU must be unique"}), 409
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": str(e)}), 500
    finally:
        db.session.close()


@app.route('/api/inventory', methods=['POST'])
def add_inventory():
    try:
        # Validate request data
        data = inventory_schema.load(request.json or {})
    except ValidationError as err:
        return jsonify({"errors": err.messages}), 400

    try:
        # Check if warehouse exists
        warehouse = db.session.get(Warehouse, data["warehouse_id"])
        if not warehouse:
            return jsonify({"error": "Warehouse not found"}), 404

        # Check product exists
        product = db.session.get(Product, data["product_id"])
        if not product:
            return jsonify({"error": "Product not found"}), 404

        # Check if inventory exists for product-warehouse
        inventory = Inventory.query.filter_by(
            product_id=data["product_id"],
            warehouse_id=data["warehouse_id"]
        ).first()

        if inventory:
            inventory.quantity += data["quantity"]
        else:
            inventory = Inventory(
                product_id=data["product_id"],
                warehouse_id=data["warehouse_id"],
                quantity=data["quantity"],
                threshold=data.get("threshold", 0)
            )
```

```
            db.session.add(inventory)

        db.session.commit()

        return jsonify({
            "status": 201,
            "message": "Inventory created / updated successfully",
            "inventory_id": inventory.id
        }), 201

    except Exception as e:
        db.session.rollback()
        return jsonify({"error": str(e)}), 500
    finally:
        db.session.close()
```

Add product Schema

```
class AddProductSchema(Schema):
    name = fields.String(required=True, validate=Length(min=3, max=100))
    sku = fields.String(required=True, validate=Length(min=5, max=100))
    price = fields.Decimal(required=True, as_string=True)
    description = fields.String(required=False)
```

Add Inventory Schema

```
class AddInventorySchema(Schema):
    product_id = fields.String(required=True)
    warehouse_id = fields.String(required=True)
    quantity = fields.Integer(required=True, validate=Range(min=0, max=100000))
    threshold = fields.Integer(required=False, validate=Range(min=0, max=10000))
```