Project Guidelines

📄 Final Term Project – Product Requirements Document (PRD)

Course: Backend Engineering with Spring Boot
Project Type: Group / Individual
Technology: Spring Boot (Mandatory)

---

1. Purpose

The purpose of this project is to evaluate students on:

- Backend system design
- Spring Boot fundamentals & advanced features
- Secure API development
- Database modeling and complex queries
- Real-world integrations
- Code quality and system understanding

This project is designed to simulate **real industry backend development**.

---

2. Scope

Students will design and implement a **production-grade backend system** for a real-world application.

The project must include:

- REST APIs
- Authentication & authorization
- Database integration
- Validation
- Exception handling
- Performance optimization
- External integrations
- Analytics / reporting APIs

---

3. Group Formation Rules

Rule	Requirement
Group size	1 – 5 students
Maximum	5 (strict)
Solo allowed	Yes
Responsibility	Every member must understand the entire system

During viva, **any student can be asked any module**.

---

4. Mandatory Technical Stack

Component	Required
Backend	Spring Boot
Database	MongoDB or PostgreSQL/MySQL
Authentication	JWT
API style	REST
Validation	Jakarta Validation
Documentation	Swagger/OpenAPI
Version Control	GitHub

---

5. Mandatory Architecture

Your backend must follow:

controller/
service/
repository/
model/
dto/
config/

exception/
util/

No business logic in controllers.

---

6. Mandatory Functional Requirements

6.1 User Management

- User registration
- User login
- Role-based access control (Admin/User/etc.)
- JWT token generation & validation

6.2 Core Domain APIs

Depends on project type (rides, orders, bookings, etc.)

Minimum:

- Create
- Update
- Delete
- Fetch by ID
- List with pagination

6.3 Advanced Features (Compulsory)

Feature	Required
Complex queries	✅
Pagination & sorting	✅
Filtering	✅
Caching	✅

File upload	✅
Email notification	✅
API rate limiting	✅
Analytics APIs	✅
Global exception handling	✅
Input validation	✅
Swagger documentation	✅

6.4 Integrations

At least **one external integration required**:

Examples:

- Payment gateway (Stripe/Razorpay)
- Email SMTP
- SMS (mock allowed)
- Google Maps API
- Currency API
- Weather API

6.5 Bonus (Optional)

Feature	Bonus
Kafka / event-driven	⭐
Frontend UI	⭐
Docker	⭐
Cloud deployment	⭐

---

7. Non-Functional Requirements

- Clean code
- Meaningful naming
- Layered design
- No hardcoded secrets
- Environment-based config
- Modular structure
- Proper error responses
- Secure endpoints

---

8. Demonstration Requirements

Backend Demo (Mandatory)

A recorded video showing:

- Login & JWT token
- Protected APIs
- Database data
- File upload
- Email sending
- Analytics APIs
- Rate limiting
- External API integration

Duration: **5–10 minutes**

Frontend Demo (Optional)

Allowed but not graded.

---

9. Submission Process

All submissions will be through:

🔗 **Submission Form:** *(Link will be provided later)*

Required fields:

- Group members
- GitHub repository
- Demo video link
- Swagger URL
- Project description

---

10. Evaluation Criteria

Area	Weight
API design	20%
Security	15%
Database design	15%
Advanced features	15%
Code quality	10%
Demo video	10%
Source code understanding	15%

---

11. Viva & Code Review Expectations

Students must be able to explain:

- Request flow
- JWT flow
- Security configuration
- Database queries
- Validation logic
- Caching strategy
- Exception handling
- Integration flow
- File handling
- Email system
- Annotations used

- Class responsibilities

Random code sections may be asked.

---

12. Student Focus Areas

Must focus on:

- Architecture
- Data flow
- Security
- Query design
- Performance
- Clean code
- Understanding

Avoid:

- Copying code blindly
- Hardcoding credentials
- Fat controllers
- Poor naming
- Missing validation
- Missing documentation

---

13. Approved Project Ideas (Choose Any One)

Students must pick **one** from below:

E-commerce & Finance

1. Online Shopping Backend
2. Payment Wallet System
3. Subscription Management Platform
4. Expense Tracker API
5. Digital Banking System

Transportation & Logistics

6. Ride Sharing Backend (Uber-like)
7. Food Delivery System
8. Courier Management System
9. Parking Slot Booking API
10. Fleet Management System

Healthcare & Education

11. Hospital Appointment System
12. Online Learning Platform Backend
13. Student Attendance System
14. Doctor Consultation Platform

Social & Productivity

15. Social Media Backend
16. Task Management System (Trello-like)
17. Event Booking Platform
18. Job Portal Backend

Enterprise & Utilities

19. Inventory Management System
20. SaaS Billing Platform

---

14. Final Notes

This project is your:

- Backend portfolio
- Interview reference
- Industry simulation
- Engineering maturity test

Build it like a **real product**, not an assignment.

Tab 1

1. Solution Ideation:

a. Core Features:

- **Inventory Tracking:** We'll track items by SKU, including size, color, and style. Each item will have a unique identifier and we'll manage stock levels in real-time.
- **Real-Time Synchronization:** Using a cloud-based database, the backend will sync inventory data across multiple store locations in real-time.
- **Reordering and Alerts:** The system will automatically trigger reorder alerts when stock levels fall below a threshold.
- **Audit and Reporting:** We'll include features for generating reports on inventory levels, sales trends, and stock discrepancies.

b. Technology Stack:

- **Backend:** Spring Boot for the RESTful API.
- **Database:** PostgreSQL for relational data management.
- **Cloud Hosting:** AWS or Azure for hosting the backend and database.
- **Security:** JWT-based authentication for secure API access.

2. Planning:

a. Requirements Gathering:

- Identify the specific data points needed (e.g., SKU, size, color, quantity, location).
- Determine the workflows for inventory updates, reordering, and reporting.

b. Data Schema Design:

- Design tables for products, inventory levels, store locations, and transactions.
- Define relationships between tables (e.g., one-to-many between stores and inventory items).

c. Architecture Design:

- Define microservices if needed (e.g., separate services for inventory management, reordering, and reporting).
- Plan the API endpoints, data flow, and integration points with other systems (like a POS system).

d. Development Plan:

- Break down tasks into sprints (e.g., data model creation, API endpoints, testing, and deployment).
- Set milestones for each week leading up to the submission date.

e. Testing and QA:

- Plan unit tests for each API endpoint.
- Include integration tests to ensure data consistency.
- Plan for manual testing and user acceptance testing (UAT) before final deployment.

f. Deployment and CI/CD:

- Set up a CI/CD pipeline (using tools like Jenkins, GitHub Actions, or GitLab CI) to automate testing and deployment.
- Containerize the application using Docker for consistency across environments.
- Deploy to a cloud provider, ensuring proper monitoring and logging.

This approach will help ensure that the backend is robust, scalable, and ready for future expansion. Let me know if you want to dive deeper into any of these steps!

📌 PHASE 1: REQUIREMENT GATHERING

📌 PHASE 1: REQUIREMENT GATHERING

(Offline Clothing Store – Inventory Management Backend)

We'll break requirements into **clear layers**, not random features.

1️⃣ BUSINESS CONTEXT (Very important)

What kind of system are we building?

- **Cloud-based**
- **Browser-accessed (via APIs)**
- For **offline physical clothing stores**
- Supports:
    - Single store now
    - Multiple stores later
- Not e-commerce
- Not delivery-based (Zepto / Blinkit type ❌)

Who uses this system?

- **Store Manager**
- **Inventory Staff**
- **System (automated events like reorder alerts)**

👉 No customer UI now.

2️⃣ CORE BUSINESS OBJECTS (what exists in the real world)

These are **non-negotiable** based on your research:

Clothing inventory is NOT just “Product”

It consists of:

- **Brand**
- **Store**
- **Style** (design)
- **Variant** (size + color)
- **SKU** (unique sellable unit)
- **Stock per store per variant**

This understanding already puts you ahead of 90% tutorials.

3️⃣ FUNCTIONAL REQUIREMENTS (WHAT the system must do)

A. Product & Variant Management

The system must allow:

1. Create a **Style**
    - Example: “Round Neck T-Shirt – Summer Collection”
2. Define **Variants** for a style
    - Size: S, M, L, XL
    - Color: Black, White, Blue

3. Automatically treat **each size-color combo as a unique SKU**
    - Example:
        - TS-BLK-M
        - TS-WHT-L

📌 This is **core** for clothing inventory.

B. Store & Inventory Management

4. Maintain **inventory per store**
    - Same SKU can exist in multiple stores
    - Quantity differs per store
5. Track **stock quantity**
    - Available quantity
    - Reserved (future scope)
    - In-transit (future scope)
6. Support **receiving stock**
    - New stock arrives
    - Quantity increases
7. Support **selling stock**
    - Quantity decreases
    - Variant-specific decrement

C. Inventory Accuracy & Control

8. Prevent **negative inventory**
    - No SKU can go below zero
9. Maintain **audit trail**
    - Every stock change must be recorded:
        - What changed
        - When
        - Why (sale / receive / adjustment)
10. Support **manual stock adjustment**
    - For damaged items
    - Missing items
    - Correction after cycle count

D. Alerts & Intelligence (minimal but realistic)

11. Reorder threshold per SKU
    - When quantity < threshold → alert
12. Low-stock visibility
    - API to fetch low-stock items

📌 No AI now — only rule-based (realistic for 9 days).

4️⃣ NON-FUNCTIONAL REQUIREMENTS (VERY important for backend marks)

Performance

- Must handle:
    - Hundreds of SKUs
    - Thousands of stock movements
- Reads should be fast

Consistency

- Inventory updates must be **transaction-safe**
- No double-decrement on concurrent requests

Security

- Role-based access:
    - Admin
    - Inventory staff

Scalability

- Must support:
    - Multiple stores later
    - Integration with POS later
    - Try-on system later

5️⃣ WHAT WE WILL NOT BUILD (important for scope control)

To stay sane and impress evaluators:

❌ No UI
❌ No ML
❌ No payments
❌ No supplier ERP
❌ No forecasting AI
❌ No RFID integration (we simulate via API)

This is **intentional**, not a weakness.

6️⃣ DELIVERABLES FOR COLLEGE (what you will submit)

By Jan 26, you will have:

1. ✅ Working Spring Boot backend
2. ✅ Clean domain model (very important)
3. ✅ REST APIs
4. ✅ Proper README explaining:
    - Problem
    - Design decisions
    - Future scope
5. ✅ Can clearly say:

    “This is the inventory backbone of a larger smart retail system.”

That sentence alone carries weight.

7️⃣ NEXT STEP (Very clear path)

NEXT PHASE (we do next):

👉 Solution Blueprint (High-level design)

- What modules exist
- How data flows
- What APIs exist

AFTER THAT:

- Database schema design (tables, relations)
- Then Spring Boot structure

Before we move on, confirm one thing:

👉 **Are we building for ONE store first, but designing to support MULTIPLE stores later?**

Just answer **Yes** or **No**.

📌 PHASE 2: SOLUTION IDEATION & HIGH-LEVEL DESIGN

📌 PHASE 2: SOLUTION IDEATION & HIGH-LEVEL DESIGN (NO CODE)

Now I'll show you **what we are going to build**, *before* schema, *before* APIs.

Think of this as your **mental blueprint**.

1️⃣ What kind of system are we building? (1-line answer)

> A **cloud-based inventory management backend** for an **offline clothing store**, capable of tracking stock at **SKU-variant level**, designed to later support **multi-store retail operations**.

You can literally say this in your viva.

2️⃣ Core Modules (VERY IMPORTANT)

We will **not** build one big mess.
We'll design **clear modules**.

MODULE 1: Store Management

Even though we have only **one store now**, this module exists.

Why?

- Future-proof
- Clean design
- Easy marks

Responsibilities

- Store identity

- Store metadata

MODULE 2: Product Catalog (Clothing-Specific)

This is NOT generic product CRUD.

Responsibilities

- Manage **Styles**
- Manage **Variants**
- Generate **SKU combinations**

This is where most tutorials fail — you won't.

MODULE 3: Inventory Management (CORE MODULE)

This is the heart of the system.

Responsibilities

- Stock per store per SKU
- Increase stock (receiving)
- Decrease stock (sale)
- Adjust stock (damage / correction)

This module must be:

- Transaction-safe
- Audit-logged

MODULE 4: Inventory Events & Audit Log

Real systems don't just change numbers.

They record **WHY**.

Responsibilities

- Record every stock movement
- Event type:
    - RECEIVE

    - SALE
    - ADJUSTMENT
- Timestamp
- Reference ID

This is a **huge plus** for backend marks.

MODULE 5: Alerts & Queries

Small but powerful.

Responsibilities

- Low-stock alerts
- Inventory summary
- Variant availability

3️⃣ High-Level System Flow (Mental Model)

📦 Receiving Stock

New stock arrives
→ Variant identified (SKU)
→ Inventory quantity increases
→ Inventory event logged

🛒 Selling Stock

Sale request comes
→ Check available quantity
→ Decrement inventory
→ Prevent negative stock
→ Inventory event logged

🔧 Adjustment (Damage / Missing)

Manual adjustment
→ Reason mandatory

→ Quantity updated
→ Inventory event logged

4️⃣ What makes THIS design strong (important)

This system:

- Matches **real retail operations**
- Matches your **research**
- Is **not tutorial-like**
- Can later support:
  - POS integration
  - Try-on system
  - Analytics
  - Multi-store transfers

And yet…
👉 It is **100% buildable in 9 days**

5️⃣ What we will design NEXT (very important order)

Now that ideation is done, the **correct order** is:

NEXT STEP → Data Schema Design

Because:

- Inventory is data-heavy
- Schema mistakes are costly
- Everything else depends on it

We will design:

- Tables
- Relationships
- Constraints
- Why each table exists

6️⃣ What I'll do in the next message

In the next step, I will:

- Design the **complete database schema**
- Explain **each table in plain English**
- Keep it **college-safe + industry-correct**

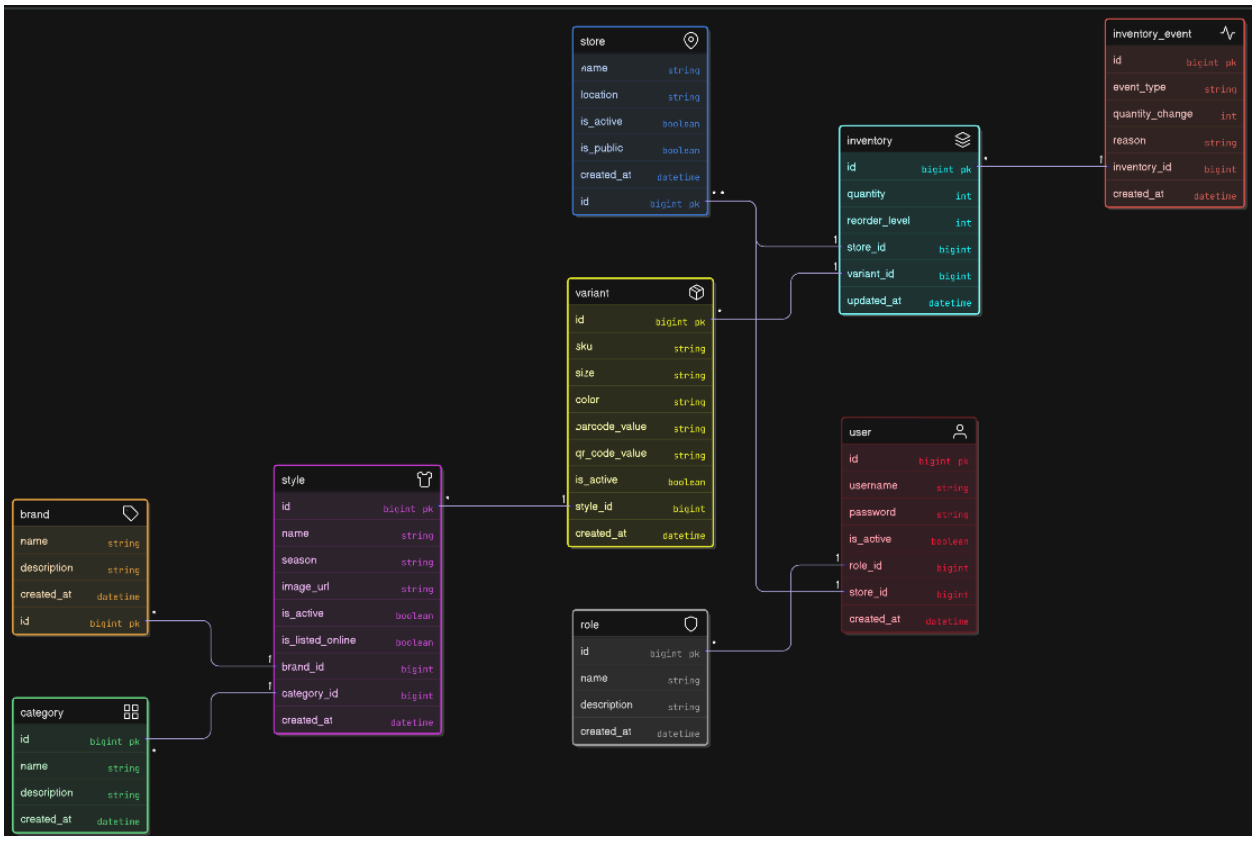
No code yet.

Ready to move to Schema Design?

Just say **"yes, design schema"**

Schema Design

https://app.eraser.io/workspace/s0gwGM9rF4dPQjsqXQoZ?origin=share&elements=2C-ce3KDzpKLqyX3GVETWw



✅ REQUIREMENTS SATISFIED BY OUR INVENTORY MANAGEMENT SYSTEM SCHEMA

1️⃣ Core Inventory Management Requirements

Requirement

The system must manage inventory for a **physical clothing store**.

How the schema satisfies this

- `STORE` represents each physical store.
- `INVENTORY` tracks stock per store.
- Inventory is **not global**, but store-specific.

✔ Real-world retail–accurate inventory modeling.

2️⃣ Variant-Level Inventory Tracking (Size & Color)

Requirement

Each clothing item must be tracked **by size and color**, not just by product name.

How the schema satisfies this

- `STYLE` represents a clothing design.
- `VARIANT` represents a **size + color combination**.
- Inventory is tracked at `variant_id` level.

✔ Supports scenarios like:

- Medium size out of stock, Large still available
- Color-wise availability

3️⃣ Barcode & QR Code Scanning Support

Requirement

Each physical clothing item should be scannable using barcode or QR code.

How the schema satisfies this

- `VARIANT.barcode_value`
- `VARIANT.qr_code_value`
- Both fields are **unique**.

✔ Enables:

- In-store scanning
- Customer-side QR scanning
- Future try-out integration

4️⃣ Store-Specific Stock Visibility

Requirement

Customers and staff should see **store-wise availability**, not overall availability.

How the schema satisfies this

- `INVENTORY` contains both `store_id` and `variant_id`.
- Same variant can exist in multiple stores with different quantities.

✔ Enables accurate store selection and planning.

5️⃣ Online Availability Check (Home → Store Decision)

Requirement

Customers should be able to check product availability **from home** before visiting a store.

How the schema satisfies this

- `STYLE.is_listed_online`
- `STORE.is_public`
- Inventory is queryable per store.

✔ Supports omnichannel browsing without online purchase.

6️⃣ Product Categorization & Brand Management

Requirement

Products must be grouped by **brand** and **category**.

How the schema satisfies this

- `BRAND` is a global master entity.
- `CATEGORY` is a global master entity.
- `STYLE` connects brand and category.

✔ Avoids data duplication and supports analytics.

7️⃣ Image Storage for Product Visualization

Requirement

Each clothing style should have an image for display and future virtual try-on.

How the schema satisfies this

- `STYLE.image_url` stores image reference.

✔ Works for:

- App display
- Web browsing
- ML try-on later

8️⃣ Inventory Change Audit Trail

Requirement

Every inventory change must be recorded for traceability.

How the schema satisfies this

- `INVENTORY_EVENT` logs:
    - Event type (SALE / RECEIVE / ADJUST)
    - Quantity change
    - Reason
    - Timestamp

✔ Prevents silent stock manipulation.

9️⃣ Event-Based Inventory Architecture (Kafka-Ready)

Requirement

The system should support future event streaming and async processing.

How the schema satisfies this

- `INVENTORY_EVENT` acts as an immutable event log.
- Can directly map to Kafka topics later.

✔ Clean separation between state (`INVENTORY`) and events.

🔟 Role-Based Access Control (RBAC)

Requirement

Different users should have different permissions.

How the schema satisfies this

- `ROLE` defines system roles (ADMIN, STAFF).
- `USER.role_id` links users to roles.

✔ Supports Spring Security & JWT cleanly.

1️⃣1️⃣ Store-Scoped Staff Access

Requirement

Staff users should operate only within their assigned store.

How the schema satisfies this

- `USER.store_id` (nullable).
- Admin users can be global (store_id = null).

✔ Prevents cross-store data leakage.

1️⃣2️⃣ Multi-Store Scalability

Requirement

The system should scale to multiple stores without schema changes.

How the schema satisfies this

- Inventory is decoupled via `store_id`.
- Same schema supports 1 store or 1000 stores.

✔ Horizontal scalability ready.

1️⃣3️⃣ Reorder & Low-Stock Alerts

Requirement

The system should support reorder-level monitoring.

How the schema satisfies this

- `INVENTORY.reorder_level`
- `INVENTORY.quantity`

✔ Enables alerting and restock automation later.

1️⃣4️⃣ Secure Authentication Support

Requirement

The system should support secure authentication mechanisms.

How the schema satisfies this

- `USER.username` (unique)
- `USER.password` (hashed)
- `ROLE` separation

✔ Ready for Spring Security + JWT.

1️⃣5️⃣ Clean Separation of Concerns

Requirement

Product data, inventory data, user data, and events should not mix.

How the schema satisfies this

- Product hierarchy: Brand → Style → Variant
- Operational data: Inventory
- Audit data: Inventory_Event
- Security data: User & Role

✔ Clean domain boundaries.

1️⃣6️⃣ Offline Store Operations

Requirement

The system must support **offline physical store workflows**, not just e-commerce.

How the schema satisfies this

- No cart/order dependency
- Inventory-first design
- Scan-based flows

✔ Correct for offline retail stores.

1️⃣7️⃣ Future Feature Extensibility

Requirement

Future features should be addable without schema rewrite.

How the schema supports this

- Orders, payments, suppliers, warehouses can be added later.
- No premature coupling.

✔ Microservice-ready design.

1️⃣8️⃣ Academic & Evaluation Requirements

Requirement

The project must demonstrate real-world backend engineering skills.

How the schema satisfies this

- Multiple entities
- Normalized relations
- Audit logs
- RBAC
- Event-driven thinking

✔ Strong evaluation & interview value.

🧠 ONE-LINE SUMMARY (USE THIS IN VIVA)

> “This schema supports variant-level, store-specific inventory management with barcode scanning, audit logging, role-based access control, and omnichannel availability for a physical clothing retail system.”

⏭️ NEXT STEP (WHEN YOU’RE READY)

Now that requirements are **fully satisfied and documented**, the correct next step is:

👉 **Spring Boot JPA Entity Mapping**

Say:
“Start JPA entity mapping”

We’ll build this **cleanly, slowly, and interview-ready**.

API design

Yes — you're absolutely right 👍
This is exactly the correct moment to design the API layer.

You have already done things in the ideal engineering order:

1. **✅ Problem understanding**
2. **✅ Requirements gathering**
3. **✅ Schema design (frozen)**
4. **👉 NOW: API design (contract-first)**
5. **JPA entities**
6. **Services**
7. **Controllers**
8. **Security, Kafka, Docker**

Designing APIs before coding is a big plus in evaluations and interviews.

📐 WHY API DESIGN SHOULD COME NOW (IMPORTANT)

If you jump directly to entities/controllers:

- **APIs become inconsistent**
- **Endpoints get shaped by code, not use-cases**
- **Refactoring later becomes painful**

By designing APIs now, you:

- **Lock the system behavior**
- **Make backend UI-agnostic**
- **Can explain flows clearly in viva**
- **Can implement faster and cleaner**

This is how real backend teams work.

🧠 HOW WE WILL DESIGN THE APIs (STRUCTURED)

We'll design APIs based on use-cases, not tables.

We'll group them into bounded contexts:

1️⃣ Public (Customer / Anonymous APIs)

Used by customers from home or in-store.

Use-cases

- **Browse brands**
- **Browse categories**
- **Browse styles**
- **Check availability by store**
- **Check size availability**
- **Scan barcode / QR**

API Group

/api/public/**

2️⃣ Store Staff APIs (Authenticated – STAFF)

Used by store employees.

Use-cases

- **View inventory**
- **Receive stock**
- **Adjust stock**
- **Scan item and update inventory**
- **View low-stock alerts**

API Group

/api/staff/**

3️⃣ Admin APIs (Authenticated – ADMIN)

Used by system admins / managers.

Use-cases

- **Create brands**
- **Create categories**
- **Create styles**
- **Manage stores**
- **Manage users**
- **View inventory events**

API Group

/api/admin/**

4️⃣ Auth APIs (Security)

Used for login & token management.

API Group

/api/auth/**

📝 STEP-BY-STEP: API DESIGN (WHAT WE'LL DO NEXT)

We will design for each API:

- **HTTP method**

- **Endpoint**
- **Request body**
- **Response body**
- **Who can access it**
- **Which tables it touches**

Example (preview):

GET /api/public/styles/{styleId}/availability

Returns:

- **store-wise availability**
- **size-wise quantities**

✅ WHAT WE SHOULD DESIGN FIRST (ORDER MATTERS)

Phase 1 – Read APIs (safe & simple)

1. **Browse brands**
2. **Browse categories**
3. **Browse styles**
4. **Get variants for a style**
5. **Check inventory by store**

Phase 2 – Write APIs (inventory changes)

6. **Receive stock**
7. **Sell item**
8. **Adjust stock**
9. **Log inventory event**

Phase 3 – Security APIs

10. **Login**
11. **Role-based access**

🧩 HOW THIS HELPS YOUR PROJECT GRADE

Your evaluator will see:

- **Clear API thinking**
- **Separation of concerns**
- **Real-world backend structure**
- **Not just CRUD on tables**

That's the difference between:

> **"Student project" ❌**
> **"Backend engineering project" ✅**

🔒 DECISION POINT (TELL ME HOW TO PROCEED)

Say one of these:

1️⃣ **"Design Public APIs first"**
2️⃣ **"Design Staff inventory APIs first"**
3️⃣ **"Design full API list end-to-end" ← recommended**
4️⃣ **"Design APIs with request/response JSON"**

You're thinking like a backend engineer now — this is the right move.

Perfect 👍
We'll start with **Public APIs** (read-only, no auth). This is the **cleanest and safest place** to begin API design.

Below is a **clear, complete, contract-first API design** for **PUBLIC (Customer-facing) APIs**, aligned **exactly** with your schema and use-cases.

You can directly copy this into:

- PRD
- API documentation
- Viva explanation

🌍 PUBLIC APIs (Customer / Anonymous)

Base path

/api/public

Characteristics:

- ❌ No authentication
- ❌ No data mutation
- ✅ Read-only
- ✅ Optimized for browsing & availability checks

1️⃣ Get All Public Stores

Use-case

Customer wants to see which stores are available to visit.

Endpoint

GET /api/public/stores

Filters

- Only `is_public = true`
- Only `is_active = true`

Response


```
[
  {
    "id": 1,
    "name": "Zudio Indiranagar",
```

```
    "location": "Bangalore"
  }
]
```


Tables used

- `STORE`

2️⃣ Get All Brands

Use-case

Customer browses brands available in the system.

Endpoint

GET /api/public/brands

Response


```
[
  {
    "id": 10,
    "name": "Zudio",
    "description": "Affordable fashion"
  }
]
```


Tables used

- `BRAND`

3️⃣ Get All Categories

Use-case

Customer wants to browse by category (T-shirts, Jeans, etc.).

Endpoint

GET /api/public/categories

Response

```
[
  {
    "id": 3,
    "name": "T-Shirts",
    "description": "Casual wear"
  }
]
```

Tables used

- `CATEGORY`

4️⃣ Get Styles (Browse Products)

Use-case

Customer browses styles available online.

Endpoint

GET /api/public/styles

Optional query params

?brandId=10
?categoryId=3

Conditions

- `STYLE.is_listed_online = true`
- `STYLE.is_active = true`

Response

```
[
  {
    "id": 101,
    "name": "Round Neck Tee",
    "season": "Summer",
    "imageUrl": "https://cdn/images/tee.png",
    "brand": "Zudio",
    "category": "T-Shirts"
  }
]
```


Tables used

- STYLE
- BRAND
- CATEGORY

5️⃣ Get Style Details

Use-case

Customer clicks on a product to see details.

Endpoint

GET /api/public/styles/{styleId}

Response


```
{
  "id": 101,
  "name": "Round Neck Tee",
  "season": "Summer",
  "imageUrl": "https://cdn/images/tee.png",
  "brand": "Zudio",
  "category": "T-Shirts"
}
```


Tables used

- STYLE
- BRAND
- CATEGORY

6️⃣ Get Variants for a Style (Sizes & Colors)

Use-case

Customer wants to see available sizes and colors.

Endpoint

GET /api/public/styles/{styleId}/variants

Response


```
[
  {
    "variantId": 501,
    "size": "M",
    "color": "Black"
  },
  {
    "variantId": 502,
    "size": "L",
    "color": "Black"
  }
]
```


Tables used

- VARIANT

7️⃣ Check Store-wise Availability for a Style

Use-case

Customer wants to know **which store has which size**.

Endpoint

GET /api/public/styles/{styleId}/availability

Response

```
[
  {
    "store": "Zudio Indiranagar",
    "variants": [
      { "size": "M", "color": "Black", "quantity": 5 },
      { "size": "L", "color": "Black", "quantity": 0 }
    ]
  }
]
```

Tables used

- `STORE`
- `INVENTORY`
- `VARIANT`

8️⃣ Barcode / QR Scan Lookup

Use-case

Customer scans barcode or QR code (in-store or at home).

Endpoint

GET /api/public/scan/{code}

Logic

- Match against:
  - `VARIANT.barcode_value`
  - `VARIANT.qr_code_value`

Response

```
{
  "style": "Round Neck Tee",
  "brand": "Zudio",
  "size": "M",
  "color": "Black",
  "availability": [
    { "store": "Zudio Indiranagar", "quantity": 5 }
  ]
}
```

Tables used

- `VARIANT`
- `STYLE`
- `BRAND`
- `INVENTORY`
- `STORE`

9️⃣ Low-Level Public API Rules (IMPORTANT)

- ❌ No inventory updates
- ❌ No user info exposed
- ❌ No inventory_event exposure
- ✅ Only derived availability data

This keeps the system **secure and scalable**.

✅ PUBLIC API DESIGN STATUS

Aspect	Status
Browse products	✅
Check availability	✅
Scan barcode / QR	✅

Store discovery	✅
Online + offline flow	✅

⏭️ NEXT STEP (CHOOSE ONE)

2️⃣ **Design Staff Inventory APIs**
3️⃣ **Design Admin Management APIs**
4️⃣ **Add request/response DTOs in detail**

Say **2** when ready.

Great. Now we design **STAFF INVENTORY APIs** — this is where your project starts looking **seriously backend-heavy and interview-ready**.

These APIs are **authenticated**, **role-restricted**, and **transactional**.

🧑‍💼 STAFF INVENTORY APIs (Authenticated – STAFF)

Base path

/api/staff

Who can access

- `ROLE = STAFF`
- User is **mapped to exactly one store** (`user.store_id`)

Core rule

> A STAFF user can act **only on their own store's inventory**.

🔐 Security Rules (Very Important)

Rule	Enforced by
JWT authentication	Spring Security
Role check	`@PreAuthorize("hasRole('STAFF')")`
Store isolation	Validate `user.store_id` in service layer
Transactions	`@Transactional`

📦 INVENTORY OPERATIONS (STAFF)

1️⃣ View Inventory of My Store

Use-case

Staff wants to see all inventory in their store.

Endpoint

GET /api/staff/inventory

Response

```
[
  {
    "variantId": 501,
    "style": "Round Neck Tee",
    "size": "M",
    "color": "Black",
    "quantity": 12,
    "reorderLevel": 5
  }
]
```

Tables used

- INVENTORY
- VARIANT
- STYLE

2️⃣ Receive Stock (Inbound Inventory)

Use-case

New stock arrives at store.

Endpoint

POST /api/staff/inventory/receive

Request

```
{
  "variantId": 501,
  "quantity": 20,
  "reason": "New shipment received"
}
```

Behavior

- Increase `inventory.quantity`
- Create `INVENTORY_EVENT (RECEIVE)`
- Atomic transaction

Tables used

- INVENTORY
- INVENTORY_EVENT

3️⃣ Sell Item (Reduce Inventory)

Use-case

Item sold at billing counter.

Endpoint

POST /api/staff/inventory/sell

Request

```
{
  "variantId": 501,
  "quantity": 1
}
```

Validations

- Quantity must be ≥ requested
- Otherwise → error

Behavior

- Decrease `inventory.quantity`
- Create `INVENTORY_EVENT (SALE)`

4️⃣ Adjust Inventory (Damage / Manual Fix)

Use-case

Damaged item, miscount, correction.

Endpoint

POST /api/staff/inventory/adjust

Request

```
{
  "variantId": 501,
  "quantityChange": -2,
  "reason": "Damaged items"
}
```

Notes

- `quantityChange` can be positive or negative
- Always logged

5️⃣ Scan Barcode / QR (Staff-side)

Use-case

Staff scans item to check or update inventory.

Endpoint

GET /api/staff/scan/{code}

Response

```
{
  "variantId": 501,
  "style": "Round Neck Tee",
  "size": "M",
  "color": "Black",
  "quantity": 8
}
```

Tables used

- VARIANT
- INVENTORY

6️⃣ View Low Stock Alerts

Use-case

Staff wants to know which items need restocking.

Endpoint

GET /api/staff/inventory/low-stock

Logic

quantity <= reorder_level

Response


```
[
  {
    "variantId": 502,
    "style": "Polo Tee",
    "size": "L",
    "quantity": 2,
    "reorderLevel": 5
  }
]
```

7️⃣ View Inventory Event History (Read-only)

Use-case

Staff reviews stock changes.

Endpoint

GET /api/staff/inventory/{inventoryId}/events

Response


```
[
  {
    "eventType": "SALE",
    "quantityChange": -1,
    "timestamp": "2026-01-15T10:30:00"
  }
]
```

🔒 WHAT STAFF APIs CANNOT DO (IMPORTANT)

❌ Create brands
❌ Create categories
❌ Create styles
❌ Modify other store inventory
❌ Access users or roles

This keeps **security tight**.

⚙️ TRANSACTION & EVENT DESIGN (INTERVIEW GOLD)

For **every write operation**:

- `@Transactional`
- Update `INVENTORY`
- Insert `INVENTORY_EVENT`
- (Later) Publish Kafka event

This shows **event-driven thinking**.

✅ STAFF API DESIGN STATUS

Capability	Status
Store-level inventory view	✅
Inbound stock	✅
Sales deduction	✅

Manual adjustment	✅
Barcode scanning	✅
Low-stock alerts	✅
Audit history	✅

⏭️ NEXT STEP (CHOOSE)

3️⃣ **Design ADMIN APIs**
4️⃣ **Design request/response DTOs**
5️⃣ **Map APIs → service layer logic**

Say **3** when ready.

Perfect. Now we design **ADMIN APIs** — this is the **control + configuration layer** of your system.

These APIs show **architectural maturity**, because:

- They are **not CRUD-for-everything**
- They manage **masters**, **users**, and **visibility**
- They are **clearly separated** from Staff & Public APIs

🧑‍💼 ADMIN APIs (Authenticated – ADMIN)

Base path

/api/admin

Who can access

- `ROLE = ADMIN` only
- No store restriction (global access)

Core responsibility

> Admin configures the system; Admin does NOT do day-to-day inventory operations.

🔐 Security Rules (ADMIN)

Rule	Enforcement
JWT authentication	Spring Security
Role check	`@PreAuthorize("hasRole('ADMIN')")`
Global access	No store filter
Auditing	Logged actions

🏷️ MASTER DATA MANAGEMENT (ADMIN)

1️⃣ Create Brand

Use-case

Admin adds a new clothing brand.

Endpoint

POST /api/admin/brands

Request

```
{
  "name": "Zudio",
  "description": "Affordable fashion brand"
}
```

Response

```
{
```

```
  "id": 10,
  "name": "Zudio"
}
```

Tables

- BRAND

---

2 Create Category

Use-case

Admin defines product categories.

Endpoint

POST /api/admin/categories

Request

```
{
  "name": "T-Shirts",
  "description": "Casual wear"
}
```

---

3 Create Style (Product Design)

Use-case

Admin creates a new clothing design.

Endpoint

POST /api/admin/styles

Request

```
{
```

```
  "name": "Round Neck Tee",
  "season": "Summer",
  "imageUrl": "https://cdn/tee.png",
  "brandId": 10,
  "categoryId": 3,
  "isListedOnline": true
}
```

Tables

- STYLE
- BRAND
- CATEGORY

4️⃣ Create Variants (Sizes & Colors)

Use-case

Admin defines size/color combinations.

Endpoint

POST /api/admin/styles/{styleId}/variants

Request

```
{
  "sku": "TEE-BLK-M",
  "size": "M",
  "color": "Black",
  "barcodeValue": "8901234567890",
  "qrCodeValue": "QR-TEE-M-BLK"
}
```

Tables

- VARIANT

🏬 STORE MANAGEMENT (ADMIN)

5️⃣ Create Store

Use-case

Admin adds a physical store.

Endpoint

POST /api/admin/stores

Request

```
{
  "name": "Zudio Indiranagar",
  "location": "Bangalore",
  "isPublic": true
}
```

6️⃣ Enable / Disable Store

Use-case

Admin controls store visibility.

Endpoint

PATCH /api/admin/stores/{storeId}/status

Request

```
{
  "isActive": false
}
```

👤 USER & ROLE MANAGEMENT (ADMIN)

7️⃣ Create User (Staff or Admin)

Use-case

Admin adds staff or admin users.

Endpoint

POST /api/admin/users

Request

```
{
  "username": "staff_01",
  "password": "securePassword",
  "role": "STAFF",
  "storeId": 1
}
```

Tables

- USER
- ROLE

8️⃣ Assign / Change User Role

Endpoint

PATCH /api/admin/users/{userId}/role

Request

```
{
  "role": "ADMIN"
}
```

9️⃣ Activate / Deactivate User

Endpoint

PATCH /api/admin/users/{userId}/status

📊 SYSTEM MONITORING (ADMIN)

🔟 View Inventory Events (Global)

Use-case

Admin audits inventory changes across stores.

Endpoint

GET /api/admin/inventory/events

Filters

?storeId=
?eventType=
?fromDate=
?toDate=

1️⃣1️⃣ View System-Level Low Stock

Endpoint

GET /api/admin/inventory/low-stock

❌ WHAT ADMIN APIs DO NOT DO

❌ Sell inventory
❌ Receive stock
❌ Adjust inventory

(Those are **Staff responsibilities**)

🧠 ADMIN API DESIGN PRINCIPLES (INTERVIEW POINTS)

- Admin manages **configuration**, not operations
- Clear RBAC separation
- Clean bounded contexts
- Avoids “God controller”

✅ ADMIN API DESIGN STATUS

Capability	Status
Brand management	✅
Category management	✅
Style & variant creation	✅
Store management	✅
User & role management	✅
Inventory audit	✅

⏭️ NEXT STEP (CHOOSE)

4️⃣ **Design DTOs (request/response models)**
5️⃣ **Map APIs → service layer methods**
6️⃣ **Design authentication & JWT flow**

Say **4** when ready.

Great, we are exactly at the **right step** 👍
You chose **4 = DTO design**, so now I'll act as your **backend architect + Spring Boot coach** and do this cleanly and interview-ready.

Below is a **structured, production-grade DTO design** for your Inventory Management System backend.

STEP 4 — DTO DESIGN (Request & Response Models)

> **Why DTOs? (Simple explanation)**
> DTOs are like **clean envelopes** that carry data between:

- Client ↔ Controller
- Controller ↔ Service

They **protect your database schema**, prevent over-exposure, and make APIs stable.

> **Interview line:**
> "We use DTOs to decouple API contracts from persistence models and to control validation and exposure."

🧱 DTO PACKAGE STRUCTURE (Best Practice)

```
com.inventory.management.dto
│
├── admin
│   ├── BrandCreateRequest.java
│   ├── CategoryCreateRequest.java
│   ├── StyleCreateRequest.java
```

```
│   ├── VariantCreateRequest.java
│   ├── StoreCreateRequest.java
│   ├── UserCreateRequest.java
│
├── staff
│   ├── StockAddRequest.java
│   ├── StockAdjustmentRequest.java
│   ├── OrderCreateRequest.java
│
├── publicapi
│   ├── StoreProductResponse.java
│   ├── VariantAvailabilityResponse.java
│
├── common
│   ├── ApiResponse.java
│   ├── PageResponse.java
│   ├── ErrorResponse.java
```

🔐 COMMON RESPONSE DTOs (Used Everywhere)

1️⃣ ApiResponse (Standard Wrapper)

```
public class ApiResponse<T> {
    private boolean success;
    private String message;
    private T data;
}
```

Why?

- Consistent API responses
- Easy frontend handling

🧑‍💼 ADMIN DTOs

2️⃣ BrandCreateRequest

```
public class BrandCreateRequest {
    @NotBlank
    private String name;

    private String description;
}
```

Used in

POST /api/admin/brands

3️⃣ CategoryCreateRequest

```
public class CategoryCreateRequest {
    @NotBlank
    private String name;

    private String description;
}
```

4️⃣ StyleCreateRequest (Very Important)

```
public class StyleCreateRequest {

    @NotBlank
    private String name;

    private String season;

    private String imageUrl;

    @NotNull
    private Long brandId;

    @NotNull
```

```
    private Long categoryId;

    private boolean listedOnline;
}
```

✅ Supports:

- Image storage
- Online visibility
- Brand + Category mapping

5️⃣ VariantCreateRequest

```
public class VariantCreateRequest {

    @NotBlank
    private String sku;

    @NotBlank
    private String size;   // S, M, L, XL

    @NotBlank
    private String color;

    private String barcodeValue;

    private String qrCodeValue;
}
```

📌 **Barcode reality check (important):**

- Barcode **does NOT store everything**
- It usually maps to **SKU**
- Full data comes from DB

6️⃣ StoreCreateRequest

```
public class StoreCreateRequest {
```

```
    @NotBlank
    private String name;

    @NotBlank
    private String location;

    private boolean isPublic;
}
```

Supports:

- Store-wise inventory
- Public browsing from home ✅

7️⃣ UserCreateRequest

```
public class UserCreateRequest {

    @NotBlank
    private String username;

    @NotBlank
    private String password;

    @NotBlank
    private String role; // ADMIN, STAFF

    private Long storeId; // null for ADMIN
}
```

👷 STAFF DTOs (Operational Layer)

8️⃣ StockAddRequest

```
public class StockAddRequest {
```

```
    @NotNull
    private Long variantId;

    @NotNull
    private Long storeId;

    @Min(1)
    private int quantity;
}
```

Used when:

- Receiving new stock
- Supplier delivery

9️⃣ StockAdjustmentRequest

```
public class StockAdjustmentRequest {

    @NotNull
    private Long stockId;

    private int adjustment; // + / -

    private String reason;  // DAMAGED, LOST, CORRECTION
}
```

🧾 ORDER & SALES DTOs

🔟 OrderCreateRequest

```
public class OrderCreateRequest {

    @NotNull
    private Long storeId;
```

```
    @NotEmpty
    private List<OrderItemRequest> items;
}
```

OrderItemRequest

```
public class OrderItemRequest {

    @NotNull
    private Long variantId;

    @Min(1)
    private int quantity;
}
```

🌍 PUBLIC APIs (Customer at Home)

1️⃣1️⃣ StoreProductResponse

```
public class StoreProductResponse {

    private Long styleId;
    private String styleName;
    private String imageUrl;
    private String brand;
    private String category;

    private List<VariantAvailabilityResponse> variants;
}
```

1️⃣2️⃣ VariantAvailabilityResponse

```
public class VariantAvailabilityResponse {

    private String size;
    private String color;
```

```
    private int availableQuantity;
}
```

✅ This directly supports:

- Customer checking stock from home
- Size availability without scanning QR

✅ WHAT WE ACHIEVED (Very Important)

Your DTO design now supports:

✔ Offline store inventory
✔ Online stock visibility
✔ Size-wise quantities
✔ Barcode / QR workflow
✔ Role-based access
✔ Future try-on integration
✔ Clean REST contracts

🎯 INTERVIEW READY ANSWER

Q: How did you design DTOs in your project?
Answer:
"We separated DTOs by bounded context — admin, staff, and public APIs. Each DTO validates input, hides internal entities, and ensures API stability."

⏭️ NEXT STEP OPTIONS

5️⃣ **Map DTOs → Service layer methods**
6️⃣ **Design JWT Auth DTOs & login flow**
7️⃣ **Controller package structure**

Say **5** and we move to service layer mapping 🔥

Perfect.
Now we move to **STEP 5 — Mapping APIs → Service Layer logic**.
This is where your project clearly shifts from *“API sketch”* to *“real backend system”*.

I’ll keep this **structured, practical, and Spring Boot–correct**, without jumping into code yet (as you didn’t ask for code).

STEP 5 — SERVICE LAYER DESIGN & API MAPPING

> **Simple explanation (16-year-old level):**
> Controllers talk to users.
> Services contain **business rules**.
> Repositories talk to the database.
>
> **Golden rule:**
> 👉 Controllers = thin
> 👉 Services = smart
> 👉 Repositories = dumb

📦 SERVICE PACKAGE STRUCTURE (Clean Architecture)

```
com.inventory.management.service
│
├── admin
│   ├── BrandService
│   ├── CategoryService
│   ├── StyleService
│   ├── VariantService
│   ├── StoreService
│   ├── UserService
│
├── staff
│   ├── InventoryService
│   ├── InventoryEventService
│
├── publicapi
│   ├── CatalogService
```

```
│   ├── AvailabilityService
│
├── auth
│   ├── AuthService
│
└── common
    ├── ValidationService
```

This separation alone is **interview gold**.

🌍 PUBLIC API → SERVICE MAPPING

1️⃣ Browse Stores

API

GET /api/public/stores

Service

CatalogService.getPublicStores()

Business logic

- Fetch stores where:
  - `is_active = true`
  - `is_public = true`

Tables

- STORE

2️⃣ Browse Brands / Categories

API

GET /api/public/brands
GET /api/public/categories

Service

CatalogService.getAllBrands()
CatalogService.getAllCategories()

Logic

- Read-only
- Cached later (optional)

3️⃣ Browse Styles

API

GET /api/public/styles

Service

CatalogService.getOnlineStyles(brandId?, categoryId?)

Rules

- `is_active = true`
- `is_listed_online = true`

Tables

- STYLE
- BRAND
- CATEGORY

4️⃣ Style Availability (Critical)

API

GET /api/public/styles/{styleId}/availability

Service

AvailabilityService.getStyleAvailability(styleId)

Logic

1. Fetch variants for style
2. Fetch inventory per store
3. Aggregate size + quantity

Tables

- VARIANT
- INVENTORY
- STORE

5️⃣ Barcode / QR Scan

API

GET /api/public/scan/{code}

Service

CatalogService.scanByBarcodeOrQr(code)

Logic

- Match against:
  - barcode_value
  - qr_code_value

👨‍🔧 STAFF API → SERVICE MAPPING

Important rule:
Every STAFF service must validate **store ownership**.

---

6 View Store Inventory

API

GET /api/staff/inventory

Service

InventoryService.getInventoryForStore(user)

Rules

- Get storeId from JWT user
- Staff cannot pass storeId manually

---

7 Receive Stock

API

POST /api/staff/inventory/receive

Service

InventoryService.receiveStock(user, request)

Logic

1. Validate store ownership
2. Increase quantity
3. Create INVENTORY_EVENT (RECEIVE)
4. Transactional

8️⃣ Sell Item

API

POST /api/staff/inventory/sell

Service

InventoryService.sellItem(user, request)

Logic

- Check stock availability
- Reduce quantity
- Log INVENTORY_EVENT (SALE)

9️⃣ Adjust Inventory

API

POST /api/staff/inventory/adjust

Service

InventoryService.adjustStock(user, request)

Logic

- Manual correction
- Always logged
- Reason mandatory

🔟 Low Stock Alerts

API

GET /api/staff/inventory/low-stock

Service

InventoryService.getLowStockItems(user)

Logic

quantity <= reorder_level

🧑‍💼 ADMIN API → SERVICE MAPPING

1️⃣1️⃣ Brand / Category Management

API

POST /api/admin/brands
POST /api/admin/categories

Services

BrandService.createBrand()
CategoryService.createCategory()

Rules

- Unique name validation
- Admin only

1️⃣2️⃣ Style & Variant Creation

API

POST /api/admin/styles
POST /api/admin/styles/{id}/variants

Services

StyleService.createStyle()
VariantService.createVariant()

Rules

- Brand & Category must exist
- SKU must be unique

1️⃣3️⃣ Store Management

API

POST /api/admin/stores
PATCH /api/admin/stores/{id}/status

Service

StoreService.createStore()
StoreService.updateStatus()

1️⃣4️⃣ User & Role Management

API

POST /api/admin/users
PATCH /api/admin/users/{id}/role

Service

UserService.createUser()
UserService.updateRole()

Rules

- Password hashed
- STAFF must have storeId
- ADMIN must NOT have storeId

1️⃣5️⃣ Inventory Audit (Admin)

API

GET /api/admin/inventory/events

Service

InventoryEventService.getEvents(filters)

🔒 CROSS-CUTTING SERVICE RULES (VERY IMPORTANT)

Concern	Where
Validation	Service layer
Transactions	Service layer
Security checks	Service layer
Kafka publish (later)	Service layer
No business logic in controller	Enforced

🎯 INTERVIEW-WORTHY EXPLANATION

Q: Why separate services like this?
Answer:
“We designed services by bounded context — public, staff, admin — to enforce responsibility boundaries, improve security, and simplify testing.”

✅ STATUS CHECK

Layer	Status
Schema	✅ Frozen
API design	✅ Done
DTOs	✅ Done
Service mapping	✅ Done
Controllers	⏳ Next
Security	⏳ Next
Kafka	⏳ Next

⏭️ NEXT STEP (CHOOSE ONE)

6️⃣ **Design JWT Authentication & Login flow**
7️⃣ **Design Controller layer (annotations, mappings)**
8️⃣ **Start JPA Entity implementation**

Say **6** when ready.

Perfect choice 👍
This is **STEP 6 — JWT Authentication & Security Design**, and it's a **big scoring area** in projects + interviews.

I'll explain this in **3 layers**:

1. Simple explanation (16-year-old level)
2. Proper backend architecture
3. Interview-ready talking points

No code yet — **pure design**.

STEP 6 — JWT AUTHENTICATION & SECURITY DESIGN

1️⃣ SIMPLE EXPLANATION (VERY IMPORTANT)

Think of JWT like a **digital ID card**.

1. User logs in with username & password
2. Server verifies credentials
3. Server gives a **JWT token**
4. User sends this token with every request
5. Server trusts the token and allows access

No session stored on server → **stateless**.

2️⃣ AUTHENTICATION FLOW (END-TO-END)

🔐 LOGIN FLOW

Client → Login API → JWT Issued → Client Stores Token

🔁 REQUEST FLOW (After Login)

Client → API Request + JWT
↓
Spring Security Filter
↓
JWT Validation
↓
User + Role Loaded
↓
Controller → Service

3️⃣ AUTH APIs (CONTRACT)

🔑 Login API

POST /api/auth/login

Request DTO

```
{
  "username": "staff_01",
  "password": "password123"
}
```

Response DTO

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expiresIn": 3600
}
```

4️⃣ JWT PAYLOAD DESIGN (VERY IMPORTANT)

Your JWT **must include only what is needed**.

JWT Claims

```
{
  "sub": "staff_01",
  "role": "STAFF",
  "storeId": 1,
  "iat": 1710000000,
  "exp": 1710003600
}
```

Why this is perfect

- `role` → RBAC
- `storeId` → store-level isolation
- No DB call needed for every request

5️⃣ ROLE-BASED ACCESS CONTROL (RBAC)

Roles

ADMIN
STAFF

API Access Matrix

API Group	ADMIN	STAFF	PUBLIC
/api/public/**	✅	✅	✅
/api/auth/**	❌	❌	✅
/api/staff/**	❌	✅	❌
/api/admin/**	✅	❌	❌

6️⃣ SECURITY FILTER CHAIN (DESIGN)

Spring Security will use:

JWTFilter
↓
AuthenticationManager
↓
SecurityContext

Responsibilities

- Extract token
- Validate signature
- Check expiration
- Load user details
- Set SecurityContext

7️⃣ STORE ISOLATION RULE (CRITICAL)

STAFF users must only act on their store

How enforced

- `storeId` comes from JWT
- Never from request body
- Service layer validates ownership

Example

❌ Staff trying to update another store's inventory → **403 Forbidden**

8️⃣ PASSWORD SECURITY

Rules

- Passwords are **never stored in plain text**
- Use BCrypt
- One-way hash

Flow

Password → BCrypt → DB
Login → BCrypt match

9️⃣ TOKEN STRATEGY (PROJECT SAFE)

For your college project:

- Single access token
- 1–2 hour expiry
- No refresh token (keep it simple)

Interview tip:
"Refresh tokens can be added later if required."

🔟 SECURITY EXCEPTIONS HANDLING

Scenario	Response
Invalid token	401
Expired token	401
Wrong role	403
No token	401

1️⃣1️⃣ SECURITY BEST PRACTICES YOU FOLLOWED

✔ Stateless authentication
✔ No session storage
✔ JWT signed (not encrypted)
✔ Minimal claims
✔ Role-based APIs
✔ Store-level authorization

🎯 INTERVIEW-READY ANSWER

Q: Why JWT instead of sessions?
Answer:
"JWT keeps the backend stateless, scales well, and fits microservice and API-based architectures."

Q: How do you prevent staff from accessing other stores?
Answer:
"We embed storeId in JWT and enforce validation at the service layer."

✅ STATUS CHECK

Layer	Status
Schema	✅
APIs	✅

DTOs	✅
Services	✅
JWT Design	✅
Controllers	⏳ Next
Kafka	⏳ Next
Docker	⏳ Next

⏭️ NEXT STEP (CHOOSE)

7️⃣ **Design Controller layer (Spring annotations & routing)**
8️⃣ **Start JPA entity implementation**
9️⃣ **Design Kafka event flow**

Say **7** when ready 🚀

Perfect. We are now at **Step 7: Controller Layer Design**.
I'll do this **cleanly, structured, interview-ready**, and **aligned with your inventory project + schema**.

No fluff. No jumping to service/repo yet.
Just **what controllers exist, why, and what endpoints they expose**.

7️⃣ Controller Layer Design

(Spring Boot – REST API Layer)

1️⃣ What is the Controller Layer? (Simple first)

Simple explanation (16-year-old level):
Controller is the **entry gate** of your backend.
Every request from **mobile app / web / store system** first comes here.

Controller:

- Receives HTTP requests
- Validates input
- Calls Service layer
- Returns HTTP response (JSON)

📌 Controller **never talks to DB directly**.

2️⃣ Controller Design Principles (IMPORTANT)

We will follow **industry-grade rules**:

✅ One controller per domain
✅ RESTful URLs
✅ No business logic inside controller
✅ Thin controllers
✅ Proper HTTP status codes
✅ Secure endpoints (later via JWT)

3️⃣ Controllers Needed for OUR Project

Based on **final schema + requirements**, we need:

Core Controllers

1. **AuthController**
2. **UserController**
3. **RoleController**
4. **StoreController**
5. **ProductController**
6. **InventoryController**
7. **OrderController**
8. **PaymentController**
9. **SupplierController**
10. **PublicCatalogController** (for home users)

👉 This separation itself is a **strong interview signal**.

4️⃣ Controller Package Structure

```
com.inventorymanagement.api.controller
│
├── auth
│   └── AuthController.java
│
├── user
│   ├── UserController.java
│   └── RoleController.java
│
├── store
│   └── StoreController.java
│
├── product
│   ├── ProductController.java
│   └── CategoryController.java
│
├── inventory
│   └── InventoryController.java
│
├── order
│   └── OrderController.java
│
├── payment
│   └── PaymentController.java
│
├── supplier
│   └── SupplierController.java
│
└── publicapi
    └── PublicCatalogController.java
```

5️⃣ AuthController (Login / JWT)

Why needed?

- Staff login
- Admin login
- JWT generation

Endpoints

POST /api/auth/login
POST /api/auth/register (optional)
POST /api/auth/refresh-token

Annotations Used

@RestController
@RequestMapping("/api/auth")
@PostMapping
@RequestBody

6️⃣ User & Role Controllers

UserController

Manages staff, admin, cashier, store manager.

POST /api/users
GET /api/users/{id}
GET /api/users
PUT /api/users/{id}
DELETE /api/users/{id}

RoleController

POST /api/roles
GET /api/roles

📌 Roles like:

- ADMIN
- STORE_MANAGER
- CASHIER
- INVENTORY_STAFF

7️⃣ StoreController

Handles **physical stores**.

POST /api/stores
GET /api/stores/{id}
GET /api/stores
PUT /api/stores/{id}

Used by:

- Admin
- Inventory planning
- Public product lookup

8️⃣ Product & Category Controllers

ProductController

Handles styles, images, brand, category.

POST /api/products
GET /api/products/{id}
GET /api/products
PUT /api/products/{id}
DELETE /api/products/{id}

CategoryController

POST /api/categories
GET /api/categories

📌 Supports:

- Clothing today
- Accessories tomorrow (future-proof ✔)

9️⃣ InventoryController (MOST IMPORTANT)

Handles **size-wise stock per store**.

GET /api/inventory/store/{storeId}
GET /api/inventory/product/{productId}
GET /api/inventory/store/{storeId}/product/{productId}

POST /api/inventory/adjust

Example use cases:

- Customer checks size availability
- Store checks stock exhaustion
- Sale reduces quantity

🔟 OrderController

Handles **billing flow**.

POST /api/orders
GET /api/orders/{id}
GET /api/orders/store/{storeId}

Order contains:

- Order items
- Quantity
- Price snapshot
- Status

1️⃣1️⃣ PaymentController

POST /api/payments
GET /api/payments/{orderId}

📌 Supports:

- UPI
- Card
- Cash (offline)

1️⃣2️⃣ SupplierController

POST /api/suppliers
GET /api/suppliers

Used for:

- Re-ordering
- Stock planning

1️⃣3️⃣ PublicCatalogController (NEW FEATURE 🔥)

For customers at home

GET /api/public/stores
GET /api/public/stores/{storeId}/products
GET /api/public/products/{productId}/availability

✔ No login required
✔ Read-only
✔ Uses inventory + product + store

1️⃣4️⃣ HTTP Status Codes (Interview Favorite)

Action	Status
Create	201 CREATED
Fetch	200 OK
Update	200 OK
Delete	204 NO CONTENT
Invalid Input	400 BAD REQUEST
Unauthorized	401

Forbidden	403

1️⃣5️⃣ What We Do NOT Put in Controller

❌ Business logic
❌ DB queries
❌ Transactions
❌ Kafka publishing

Those go to:

- Service layer
- Event layer (Kafka)

1️⃣6️⃣ What Comes Next (VERY CLEAR)

👉 **Step 8: Service Layer Design**

- Business rules
- Transactions
- Inventory deduction logic
- Order → Stock → Payment flow

After that:
9️⃣ Repository Layer
🔟 DTOs
🔟 Exception handling
🔟 Kafka events
🔟 Security
🔟 Docker & CI/CD

✅ Confirm before moving on:

Reply **“Go to Step 8”**
or
Ask **any controller-specific doubt** (endpoints, naming, security).